A Timely Journey Through the Cloud

Millnert, Victor

2019

Document Version:
Publisher's PDF, also known as Version of record

Link to publication

Citation for published version (APA):
Millnert, V. (2019). *A Timely Journey Through the Cloud*. [Doctoral Thesis (monograph), Department of Automatic Control]. Department of Automatic Control, Lund Institute of Technology, Lund University.

Total number of authors:
1

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

A Timely Journey Through the Cloud

VICTOR MILLNERT
DEPARTMENT OF AUTOMATIC CONTROL | LUND UNIVERSITY

A Timely Journey Through the Cloud

Victor Millnert

LUNDS
UNIVERSITET

Department of Automatic Control

PhD. Thesis TFRT-1126
ISBN 978-91-7895-235-9 (print)
ISBN 978-91-7895-236-6 (web)
ISSN 0280–5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Printed in Sweden by MediaTryck.
Lund 2019

To my family.

Abstract

This thesis treats the intersection between two of the largest transformations we are seeing within our society today; the cloud and the internet-of-things (IoT). The aim of this thesis is to investigate different ways to model and control a network of cloud services so that timing-critical IoT applications can make use of them. Examples of such applications can be autonomous and mobile robots, smart production plants, or massive multi-player augmented-reality games. The main motivational use-case, however, comes from the industrial side, and their digitalization, the drive towards industrial internet-of-things (IIoT). We wish to enable smart robots to offload some of their computations to the cloud in order to allow for better and smarter control and collaboration. For instance, using the cloud, it would become possible for them to collaborate and make use of smarter analytics, artificial intelligence, and machine learning, in order to improve efficiency and safety.

To address this problem the thesis combines concepts and theory from different fields, most notably from control theory, real-time systems, and network calculus. Examples are: modeling of dynamic systems and the use of feedback and feedforward control from control theory, the goal of ensuring that end-to-end deadlines are met, from real-time systems, and finally the principles of modeling traffic from network calculus.

The thesis begins with an introduction to provide some background on cloud, IIoT, and to set the scope of the thesis. Following this, we begin by treating the problem of controlling a single cloud service with the goal of ensuring that the traffic flowing through the node is guaranteed to meet a deadline. Following this, we study a chain of connected cloud nodes, investigating how to provide end-to-end deadline guarantees for the traffic flowing through the chain. The chain is finally generalized to a network of cloud nodes, with multiple flows traversing it. For this problem we study how to ensure that the end-to-end deadline of every single flow in the network is guaranteed. We also provide a set of protocols controlling how cloud nodes and flows are allowed to dynamically join and leave the network, such that no end-to-end deadline is violated.

Acknowledgments

"A timely journey through the cloud". This title can have many interpretations, but apart from capturing the topic of this thesis, it has a quite personal meaning. This thesis is a good view into *my* journey through the cloud.

Professor Johan Eker, I could not have completed this journey without you. You have been the best mentor a PhD student could ever wish for, and I will be forever grateful for having had the opportunity to have you as my mentor. It has always been a lot of fun to work with you, and you have always pushed me and allowed me to be enthusiastic. You have patiently listened through my crazy new ideas (for instance that time I sent you a 25 p. report at 11pm on Good Friday). Whenever you thought there was room for improvement you never told me so outright, but instead asked some cryptic questions about it, letting me figure it out by myself. Thank you!

Professor Enrico Bini, I could not have done this journey without you either. You have taught me so much, from when a sub-index should be sanserif or not, to how to simplify a problem down to the most integral and important sub-components. But most of all, it has been so much fun to work with you. I will always remember all the crazy brainstorming sessions as well as the aperitivos in Italy.

Professor Karl-Erik Årzén, thank you for your never-ending support. I have always been able to count on you for help and feedback. It means so much to me that you took your time to read this thesis and give me feedback despite the fact that you were on vacation. We have been to many good workshops together, and it is always a pleasure. You have taught me the subtle art of networking and whenever the rest of us are lost, you know where to go. From the best Gin-Tonic place in Porto, Portugal, to the which direction my research should take.

Mamma and Pappa, you mean everything to me. You have always given me so much love, always believed in me, and always helped me find my own way through life. You know me better than myself (even if I will never admit this) and you have given me the best childhood and upbringing one can wish for, and I have been so lucky to have you two as my parents. Your support, in combination with your moral values and work ethic is what has truly enabled me to grow as a person and as a researcher. The proverbs may not be that many, and not always so pleasant to hear, but your ability to say them with unquestionable love has made me cherish them.

You have taught me that failure is part of life, and what matter is what we do next and how we reflect and learn from our mistakes. Here are two wonderful proverbs I one day wish to pass on to my future children: *"Early in bed, early to rise, makes a man healthy, wealthy, and wise". "Erfarenheten är den bästa läraren, men den skickar dyra räkningar".*

Ola and Mats, being your little brother is the best gift I could have asked for. You are the best role models there is. For as long as I can remember I have striven towards being like you. Although I have found my own path now, there are still many amazing things you continue to teach me and inspire me with. Seeing how you have started your own families, and to be able to share that joy with you is also something that I cherish every day. I am forever grateful that you are always there to help me, push me, and lift me up.

There are many more I would like to extend my gratitude towards. Especially Karl-Johan Åström, thank you for teaching us so much about life and research, and for always being such an inspiration. Anders Nillson, Anders Blomdell, and Leif Andersson, thank you for your patience and help will all my computer questions and problems. Eva Westin, this department would not thrive the way it does if it were not for you. And to all the fellow PhD students at this department, thank you for always being so inspiring and pushing each other. You are what makes this the best institution in the world.

To Tommy, Christian, Arian, and Lars, traveling with you was always a blast, and I am looking forward to many more conversations about the future of society, possibly over a few beers too many.

I am sure I have forgotten many names that deserve recognition, but will just say this: I could not have done this journey on my own. So to everyone who has touched my life throughout this journey, however big or small, thank you. I hope I can return the favor some day!

Financial support

Part of this work has been supported by the Swedish Research Council (VR), as well as by the LCCC Linnaeus Center, and the ELLIIT Excellence Center at Lund University. The author has also been supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

Contents

1

Introduction

It is no exaggeration that we today are seeing a technological transformation that has the potential to change our society at its very foundation. The initial seed of this transformation began to grow in the digital world of early cloud applications. These cloud applications was mostly used by other digital entities, leading to this transformation being very much a "digital only" transformation. Examples of successful applications are e-mail, online search, social media, and media streaming. After a while, the successful principles of these applications began to take root in other areas of society as well. They began to touch and interact with the physical world through a new set of applications. Examples of these are ride-sharing applications, e-commerce, smart homes/cars/toothbrushes/[insert appliance here]. The ability to gather data about physical entities, analyze it in the cloud, and make smarter decisions has the ability to transform and disrupt many industries. One example of such transformation is the ride-hailing industry. There, data of the cars and customers are constantly analyzed in order to match customers with cars and ride providers. However, it goes deeper than this, as they can continuously analyze data from their customers and find patterns, allowing them to suggest the optimal place for cars and ride-providers to "idle" (or wait for new customers). Through this they are able to minimize the overhead (meaning the time/distance cars are empty) and thus also their environmental footprint.

Where we have not yet seen the digitalization-seed take root, but will likely do so within the near future, is within traditional industries. This is one of the more challenging transformations, and therefore focus of this thesis, because it involves timing-critical applications, such as manufacturing with robots. In order to allow parts of such applications to be controlled from the cloud, the cloud must be predictable. It must be able to guarantee that a cloud service will respond within a given deadline. If the cloud service fails to operate within a timely fashion, it might lead to the application also failing, which in turn can have severe consequences. The goal of this thesis is therefore to investigate methods to ensure that cloud services can operate in a predictable and timely way:

> *"Control the cloud, so we can put control in the cloud."*

1.1 The cloud

This section introduce a brief background of cloud computing, but is by no means a comprehensive guide towards the whole of the cloud-industry. Instead, the purpose of this section is to provide you, the reader, with some basic knowledge of the history, terminology, and concepts of cloud that will be useful when reading the rest of the thesis. For an overview of the history, and possible future of the cloud industry, we refer the reader to [Armbrust et al., 2009; Jonas et al., 2019].

There are many different views of what the cloud is, and how you should define it, but in 2011, the National Institute of Standards and Technology in the USA, provided following descriptions of the key characteristics [Liu et al., 2011; Mell, Grance, et al., 2011]:

- *On-demand self-service.* A consumer can unilaterally provision computing capabilities as needed in an automatic way.

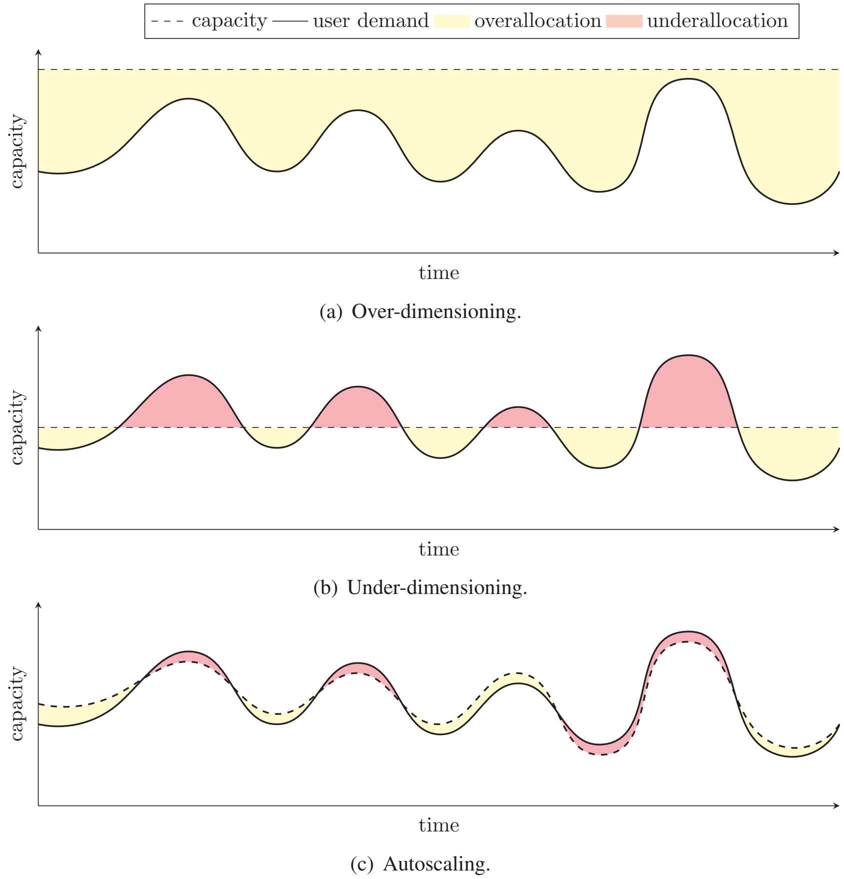
- *Broad network access.* Resources are available over the network and can be accessed through standard mechanisms.

- *Resource pooling.* The provider's computing resources are pooled to serve multiple consumers.

- *Rapid elasticity.* The compute capabilities of a customer can elastically be provisioned and released, in some cases automatically. The resources that are available to the customer appear unlimited from the perspective of the customer.

- *Measured service.* The usage by a customer can be monitored automatically, and the customer can be billed for only the amount of resources that it uses, this is often referred to as the pay-as-you-go cost model.

In the pre-cloud era, software services were commonly hosted on an on-premise datacenter. During that time, whenever the developer would need more resources, he or she would have to: i) buy/order a new physical server, ii) wait for the new server to arrive, iii) install the server, iv) set up connectivity and security between the servers, and v) deploy the software service on the server. Apart from this process taking a long time, the owners would have to pay all of the up-front cost associated with this. This led long lead-times for adjusting the computation capacity of the datacenter. One was therefore left with two options, as shown in Figure 1.1: 1) *over-dimension*, or 2) *under-dimension* the capacity of the datacenter. If they over-dimension the resources, they ensure that their service will be able to handle the peak user demand, but at a steep price. Whenever all of the resources are not used, they still pay for them. On the other hand, if they choose to under-dimension the capacity, so that they do not waste unnecessary resources, they will not be able to meet the peak



(a) Over-dimensioning.

(b) Under-dimensioning.

(c) Autoscaling.

Figure 1.1 Illustration of the concepts of over-dimensioning, under-dimensioning, and autoscaling or the capacity of a cloud service. The solid line represents the user demand for the service, and the dashed line the available capacity. The yellow region represents the amount of capacity that is wasted, and the red region the amount of capacity that is lacking.

user demand. This could potentially lead to poor customer experiences and a loss of revenue.

With great improvements in virtualization techniques, some companies were able to specialize in providing *on-demand compute resources* and the cloud-industry was born. These companies were now able to allow their customers to rent virtual machines as well as storage capabilities in a dynamic and elastic way. Through the economy of scale, by pooling their resources together, the cloud provider were also able to offer these services at competitive prices. Instead of going through the hassle of buying, waiting for, and installing new physical servers, it was now possible for the cloud customer to simply rent the computation capabilities from a cloud provides. This gave the cloud customer the illusion of near infinite computing resources. Moreover, it also eliminated the up-front cost, and instead, the cloud customer would only pay for what they used.

By moving to the cloud a third option for controlling the capacity of the software service became possible: *autoscaling*, as shown in Figure 1.1. This means that the service owner can adjust the capacity dynamically, over time, in order to match the fluctuating user demand. If done properly, this would eliminate the waste of resources which comes from over-dimensioning as well as allow the application to handle the peak user demands.

A network of microservices

The cloud computing industry has undergone an interesting journey from its early days until today. Along with improvements of virtualization and autoscaling techniques, the way that software services is hosted in the cloud has changed [Cockroft et al., 2011; Adhikari et al., 2012; Thönes, 2015]. One of these changes gave rise to a new type engineering practice, called Chaos Engineering [Chang et al., 2015; Basiri et al., 2016; Beyer et al., 2016], which involves deliberately shutting down parts of the cloud applications or cloud system, to ensure that the fault-protection mechanisms work as intended. Another change has been the way cloud application are hosted and decomposed, and is illustrated in Figure 1.2. The cloud industry has moved from hosting services in on-premise servers, to hosting them on monolithic virtual machines in the cloud, to finally hosting them as microservices in the cloud.

The services hosted on the on-premise datacenters were often *monolithic*, meaning often large, stand-alone, and self-contained applications that we can think of as silos. The application users would interact with the services, but the services would not interact amongst themselves.

During the first transition to the cloud, the developers would set up their cloud

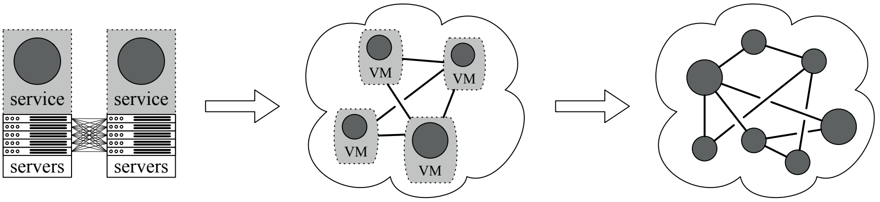


Figure 1.2 The transition from hosting a service at an on-premise datacenter, towards hosting it on a network of virtual machines in the cloud, to finally hosting it on a network of serverless functions in the cloud. From the developers perspective, this transition allowed them to shift the focus from connecting the physical servers to instead focus on connecting the services themselves.

environment by renting and connecting virtual machines, and virtual storage. This cloud-provisioning model is called *Infrastructure-as-a-Service (IaaS)*. The cloud customer would typically create a cloud environment that mimicked the one used in their previous on-premise datacenter, but with virtual machines instead of physical servers. Therefore, the structure of the software applications would still be similar to the one used before (i.e., monolithic). While this allowed for cost-savings since they only paid for what they used, it still forced the software developer to deal with many of the nitty-gritty details common when hosting software at an on-premise datacenter.

Eventually, the cloud-provisioning model offered by the cloud providers improved, and so did the practice by which software was hosted on the cloud. One shift was the development of new techniques and tools used to deploy and manage cloud-hosted software. These new tools and techniques allowed the cloud consumers to shift from large monolithic applications to a service mesh. Instead of using the large virtual machines, there was a drive to decompose them into smaller and smaller services, leading to a *network of microservices*, where each microservice had a single task and a standardized way of connecting and "talking" to the other microservices. This made it easier to develop, update, and scale the cloud application, because one could deploy, update, and scale a single microservice independently of the others.

Virtual Network Functions

Transformation of the cloud industry is rapidly expanding and affecting other industries and domains as well. One such domain is the processing of network packages, as for example found in the telecommunication industry. Network functions used to be packaged as monolithic physical appliances that were connected together using physical networks. Examples of such appliances include routers, transcoders, firewalls, switches, etc. With the improvement of the cloud technology, there is an interest to instead package these services as cloud services, and then host them in a virtual environment on standardized hardware. An initiative driven by ETSI (European Telecommunications Standards Institute) addresses the standardization of such virtual network services under the name *Network Functions Virtualization* (NFV) [ETSI, 2012]. The expected benefits from this are, among others, better hardware utilization and more flexibility, which translate into reduced capital and operating expenses [ETSI, 2013].

In Figure 1.3 a set of virtual network functions (VNFs) are illustrated, as well as the packet flows traversing the network. Each of the virtual network functions has a number of virtual resources allocated to them, and this amount can be scaled up and down independently of the other VNFs. The virtual resources are hosted on an infrastructure, called network function virtualization infrastructure (NFVI), and can be comprised of standardized servers. From the perspective of this thesis, it is especially interesting to note the similarities between the network of VNFs, and the network of microservices described in the previous section. Both of these are used

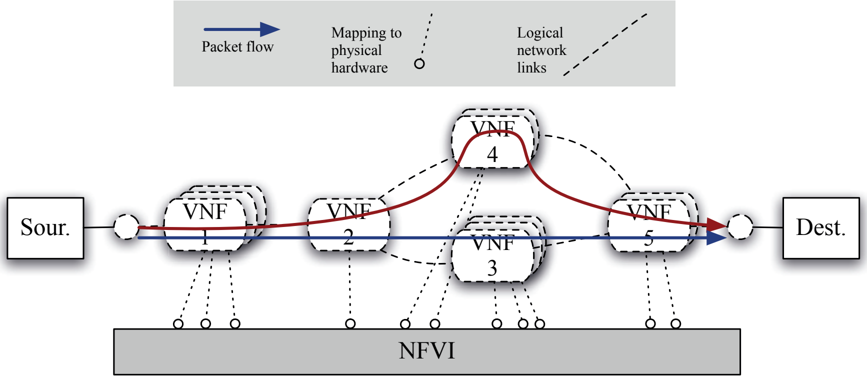


Figure 1.3 Several virtual networking functions (VNF) are connected together to provide a set of services. Each VNF consists uses of a set of virtual resources, such as virtual machine instances. In this illustration, VNF_1 consist of three virtual machines. These are mapped onto physical hardware referred to as the network function virtualization infrastructure (NFVI).

as motivational cases within this thesis, since the problem statement fits well for both of them. That is, how to control the amount of allocated virtual resources to the nodes, such that the traffic flowing through them is able to do so within a given end-to-end deadline. However, this will be discussed more thoroughly later.

1.2 Control in the cloud

Along with the improvements of the cloud computing technology, and by the ease of which computations could be offloaded to the cloud, we began to see another transformation emerge—the digitalization and the connection between the physical world and the cloud. From the perspective of a control theorist this can be described as connecting sensors and actuators to the cloud. The sensing and actuating is done in the physical world, but the computation and decisions are made in the cloud—*feedback-loop through the cloud.*

An example of one such IoT-application connected to the cloud is depicted in Figure 1.4. It illustrates an automated watering system. It has a soil sensor which sends data to the cloud, where it is analyzed. It has a smart sprinkler system (the actuator) that can be started remotely (for instance by a cloud service). Naturally, if there was only a soil sensor and a smart sprinkler there would not really be a need to send things through the cloud, because it would be simple enough to connect them and design a feedback loop that starts the sprinkler system whenever the soil is sufficiently dry. The downside, however, would be that the sprinkler might run in the morning even though the weather-forecast predicts that it will rain in the afternoon.

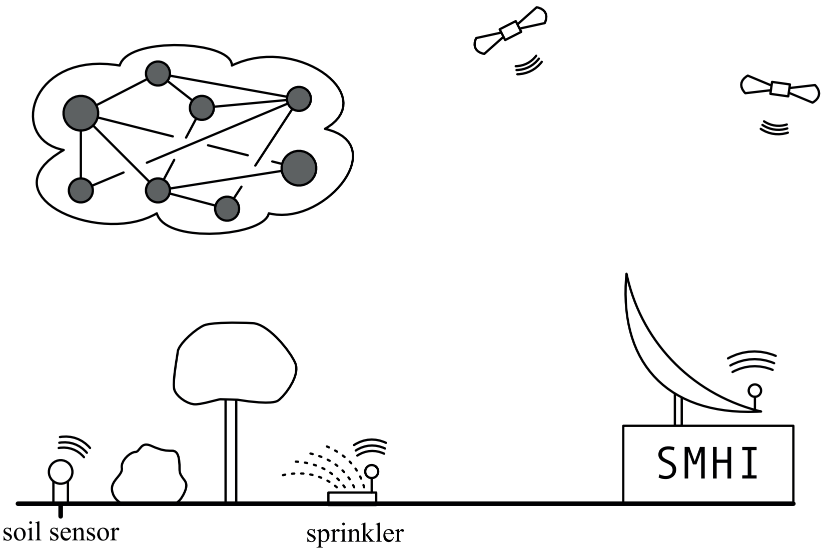


Figure 1.4 Illustration of an automatic watering system. The soil sensor sends data to the cloud. There, the data is analyzed and combined with weather forecasts from the local weather station (and the satellites). The cloud service then computes when the sprinkler system should be initiated, and how long it should run for.

Therefore, to make the system smarter, we connect it to the cloud. This will allow us to combine the soil sensor-data with other data to make a more informed decision. As an example, the soil data could be sent to the cloud where it would be processed and analyzed. Should the soil then be dry, it will call another microservice that will fetch the weather forecast from the Swedish Meteorological and Hydrological Institute (SMHI), which does weather forecasting. These two data points can then be combined with some other cloud service that fetches information about the type of plants in the garden. Together, they can figure out what the desired moisture-level of the soil should be. Therefore, if the soil needs watering, and there is not enough rain predicted, a cloud service will send a signal down to the sprinkler system telling it to run for a certain amount of time. This is a simple example, yet one that illustrate the benefit of connecting IoT-applications to the cloud. One of the main benefits is that we can allow small mobile devices to benefit from the vast computational power offered by the cloud.

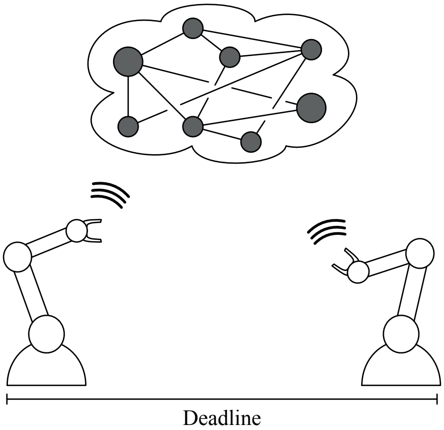


Figure 1.5 Illustration of how smart robots can be connected to the cloud, and make use of the network of microservices.

Industrial Internet of Things

There is an increasing interest from the traditional industries to also connect some of their processes to the cloud, to allow them to make use of smarter analytics and better control. An example of this is illustrated in Figure 1.5 where two industrial robots are connected to the cloud, and where part of their logic resides there. While this transformation shares some of the traits with the IoT-example described earlier (i.e., the smart sprinkler system), the industrial use-case puts more demanding requirements on the system. For instance, when two industrial robots collaborate, as illustrated in Figure 1.5, it should not take more than a predictable amount of time for the signals to go from one robot to the other. This requirement is usually expressed by establishing a deadline, i.e., it cannot take a signal more than $\mathscr{D}$ ms to reach the other robot. Should such a deadline be violated, the robots might, for example, collide. This makes this transformation towards Industry 4.0, or IIoT non-trivial, since challenges such as these has to be addressed.

Example from a Swedish manufacturer To give some motivation for the drive towards smarter manufacturing we will use an example of a Swedish manufacturer that had some issues with part of its production line [Bengtsson, 2017]. The problem was that some of their industrial robots missed their schedule some times (i.e., they would miss their deadline). To address this issue, the company contacted a top-tier consulting firm. The consulting firm did not use the cloud to attack this problem, but instead stood next to the robots (physically) and clocked them. This way, they thought, they would be able to derive a schedule for the different actions of the robots in order to find where, and why, they missed the schedules. Safe to say, their venture was unsuccessful. The consulting-firm failed, so the manufacturing company consulted a tech-firm instead. The tech-firm had the same idea: draw a schedule of the robots and figure out where the robots miss their deadlines, but they had a different angle of attack. They instead sent the program-pointer of the robots to the cloud, and analyzed it there. This way, they were able to build a system where it was possible to see exactly where in the program each robot were, in real-time. This made it easy too see what caused the issues and to address them.

The company was impressed and saw the potential if they could take this one step further–to put some of the robot-control in the cloud. At that moment, the flow was only from the robots, to the cloud, to a monitor. But instead, they would like it to be from a robot, to the cloud, and back to a robot. The only issue was that it would take too long, and be too unpredictable, to send the signals to the cloud, perform the analytics and compute the control action, and then send the control-action back to the robot.

A need for a predictable and low-latency cloud

The previous examples illustrate the need to develop a predictable and low-latency cloud, where one can guarantee that the end-to-end latency remains below an end-to-end deadline. To give some flavor of the latency needs for different applications, we have provided Figure 1.6, based on data from [Fettweis, 2014]. It illustrates that in order to have proper tactile feedback (which is required to spin a basketball on your finger), the end-to-end latency can not be more than 1 ms. For visual feedback, this is increased to 10 ms and for audio feedback (e.g., phone calls) this is further increased to 100 ms. For coarse-grained motor-functions, the end-to-end latency can be allowed to be as much as a second. Another work that has also studied the demands on the end-to-end latency is [Millnert, 2014], where the quality of a haptic-feedback controlled industrial robot was severely degraded once the round

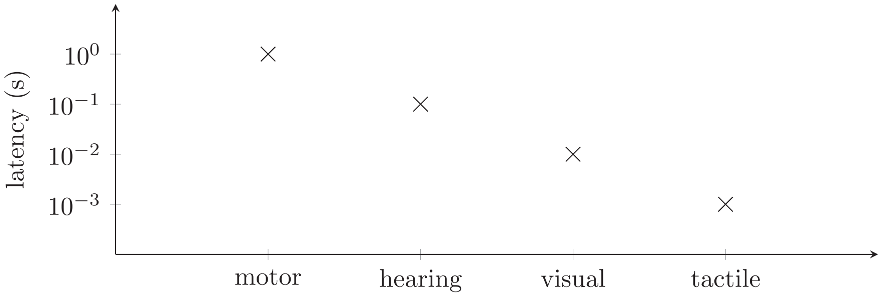


Figure 1.6 Illustration of the latency requirements for different types of feedback. When having feedback for a motor-control type system the requirement is typically a second, while for hearing feedback it is 100 ms, for visual it is 10 ms, and for tactile feedback it is as low as 1 ms.

trip-latency was larger than 20 ms.

Part of the problems with the transition towards a low-latency cloud will likely be solved by the coming 5G wireless communication technology [Simsek et al., 2016; Aijaz et al., 2016], which will enable a predictable and low-latency wireless connection to and from the cloud. What then remains is to be able to ensure that the end-to-end latency *within the cloud* remains predictable and reliable.

1.3 Controlling the cloud

As mentioned in the previous section, in order to allow industrial applications to use the cloud, it must be possible to provide a predictable end-to-end latency for the signals passing through it. The main focus of this thesis is therefore to identify a few fundamental building-blocks that can be used to ensure this in an efficient way. The goal is to illustrate how to use and combine such building blocks in different ways, and to do this we have used two tools:

- control theory, and
- network calculus.

Together they have allowed us to derive a basic model of a cloud node, illustrated in Figure 1.7, as well as the building blocks used for controlling it. In the illustration, the network traffic enters the node and is either admitted to the queue of the node, or discarded. This is controlled by the admission controller. Following this, the load balancer distributes the packets in the queue to one of the virtual machines. At any point in time, it is the job of the service controller to control how many virtual machines should be active, and processing packets. Once a packet has been processed,

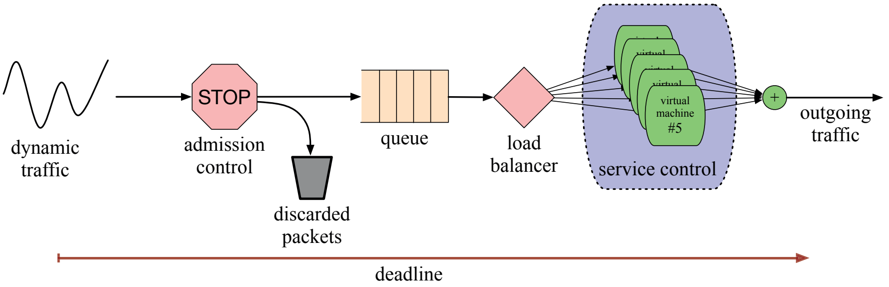


Figure 1.7 Simple illustration of a cloud node used within this thesis. Incoming requests are either admitted into a queue, or discarded–a decision made by the admission controller. From the queue, the requests are distributed to the virtual machines by the load balancer. The requests are then processed by the replicas before being forwarded to the next microservice, or sent back to the user.

it is forwarded to another cloud node (not depicted in the illustration). The goal is to ensure that all of this is done within a given deadline.

Service control The service controller is responsible for controlling the number of replicas the cloud node has active at any point in time. There are many ways of doing this, but it typically involves using some feedback from measured traffic load, the CPU-load, or the time it takes a new request to be processed. Designing an autoscaling policy can be non-trivial because:

- The latency required to add or remove computing resources.
  - A new virtual machine has to be scheduled on a physical server, the VM then needs to be initialized (boot time, mounting of file systems, etc.), and finally the application has to be initialized (load necessary libraries, synchronize with others, etc.)
- The performance of a replica is uncertain.
  - The performance is subject to what kind of physical machine it is hosted on, as well as the behavior of other virtual machines running on the same physical servers.

Admission controller Since it is difficult to exactly match the available processing capacity with the incoming demand, there might be times where there is a shortage of available processing capacity, just as in Figure 1.1(c) (the red areas). In those cases, the service might have to prioritize access for certain users and deny access to others, because it does not have the capacity to serve them all. This decision is typically made by the admission controller.

Load balancing. The responsibility of the load balancer is to distribute the incoming traffic between the replicas of the service. If this is not done in a proper way, there is a risk that some replicas will become overloaded, leading to longer response times for requests assigned to those, while other replicas might not have enough load. There are many interesting challenges and problems in this area, however, it is considered outside the scope of this thesis.

Orchestration The problem of orchestration, not illustrated in Figure 1.7, is a non-trivial problem and involves deciding how and where the virtual resources should be scheduled. For instance, it involves deciding on which physical servers the virtual machines should be running. Problems within this space are also outside the scope of this thesis.

Control theory

Control theory, and the philosophy behind it, is the bedrock of this thesis. The idea is that the cloud systems considered in this thesis should be modeled in a dynamic way. This will allow us to design control-mechanisms that will ensure that the cloud system is automatically driven into a desirable state. A dynamic model may mean different things, but considering the cloud node illustrated in Figure 1.7, it would mean that the number of virtual machines are allowed to change dynamically, over time. Taking a step back, and considering the network of microservices, it would mean that we allow for new applications and cloud nodes to dynamically join and leave the network.

The goal is therefore to capture the dynamic behavior of these systems and to design control-laws and protocols which control them in a desirable way. To do this, we will use two key concepts from control-theory: i) feedback systems, and ii) feedforward systems.

Feedback systems One way to ensure that a dynamical system is controlled in a desirable way is to use feedback. The idea is to measure a signal, and then decide on a control action such that the measured signal is driven into a desirable state. For example; in the case of a service controller this could mean that the goal is to have “just enough” virtual machines so that the latency of the cloud node is not larger than 10 ms. By introducing feedback control, the system can measure metrics such as: i) latency, ii) amount of traffic in the queue, and iii) processing capacity of the virtual machines, and then combine these to make a decision on how many virtual machines the node should have running. With this approach we can also allow for uncertainties and imperfect modeling. Uncertainties could be a virtual machine crashing, a sudden increase in the traffic, or a variation of the time it takes to process an incoming packet.

Another example where the use of feedback can be very helpful is with the admission control. Consider an example where the goal of the admission controller is to ensure that the backlog (number of packets in the queue) never exceeds the maximum queue-size. By using feedback, it is possible to guarantee this through a simple if-statement:

$$\begin{cases} \text{accept new packet} & \texttt{if} \quad \text{backlog} + \text{new packet} < \text{queue size}, \\ \text{reject new packet} & \texttt{else}. \end{cases}$$

Feedforward systems While feedback systems allow for some relaxation in how rigorous the modeling has to be, it can sometimes be too slow to react to changes. When solely using feedback for controlling a system, there has to be an error before a control action is taken. If we go back to the problem of autoscaling, having a purely reactive system might be too slow. The reason is that starting new virtual machines will take some time. Therefore, if the system sees that the latency of the cloud node reaches a critical point, it will already be too late since it will take some time to add more processing capacity and reduce the latency.

Feedforward control offers a way to allow the system to be proactive. By measuring the incoming traffic, the controller can try to predict how many virtual machines it will need in the near future, where by the near future we mean the time it takes to boot up a new VM. If it realizes that it will need some more virtual machines, it can therefore, proactively, start the extra ones.

Another way feedforward control can be used is to allow for communication between cloud nodes. For instance, if one node sees an increase in traffic, it can use feedforward control to inform other cloud nodes, which it sends traffic to, that they also will see an increase in traffic in a short while, so that they can start acting now.

Network calculus

Much of the modeling and analysis in this thesis borrows inspiration from the field of Network Calculus, since it offers a very convenient way of modeling the traffic flowing through the network, as well as simple ways to measure and compute metrics such as the backlog and latency. The theory behind network calculus is very rich, but quite new. The first two papers which lay the foundation were published by Rene Cruz at UC San Diego in 1991, [Cruz, 1991a; Cruz, 1991b]. Some other notable works on network calculus include [Le Boudec and Thiran, 2001; Jiang and Liu, 2008; Bouillard et al., 2018].

While there are many exotic and beautiful parts to network calculus, the main idea used within this thesis is their way of modeling traffic in a *cumulative way*. If we take the cloud node illustrated in Figure 1.7, we can model the rate of the arriving traffic as $r(t)$ packets per second (pps), and the rate of the departures (processed packets) as $s(t)$ pps. As illustrated in Figure 1.8, network calculus provides us with a simple way of computing the *backlog* $B(t)$ and the *latency* $L(t)$ for this system. It does so by analyzing cumulative traffic, $R(t) = \int_0^t r(x)\mathrm{d}x$ and $S(t) = \int_0^t s(x)\mathrm{d}x$. The backlog is then the *vertical distance*, and the latency is the *horizontal distance*, as illustrated in Figure 1.8. Or mathematically as:

$$B(t) = R(t) - S(t), \qquad L(t) = \inf\{\tau \geq 0 \ : \ R(t-\tau) \leq S(t)\}. \tag{1.1}$$

Providing bounds

Another useful property of network calculus is a way to provide deterministic bounds on the latency and the backlog of communication systems. This is not used within the thesis, but some ideas of how to extend this in future work is presented in Chapter 9. Instead the intuition and the ideas of how to express these bounds are used within this thesis. In network calculus, the bounds for the traffic arriving to a server is given as *arrival curves* and the bound for the processing capacity offered by a server by *service curves*. Note that within deterministic network calculus, these bounds are deterministic upper and lower bounds.

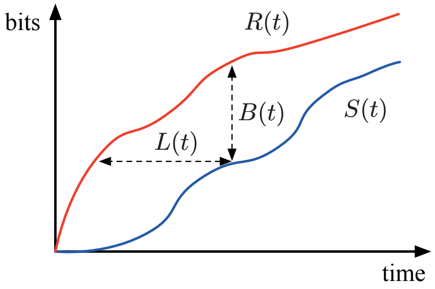


Figure 1.8 Illustration of the backlog and latency using network calculus.

Arrival Curves In network calculus it is common to provide a bound for the amount of traffic a server might see over a time period. While this is common in many different fields, what makes it unique for network calculus is that the ability to provide a bound on the *cumulative arrivals*, meaning a bound on the total amount of packets that might flow to a server during this time-period. This bound is usually denoted as an *arrival curve* α. Formally, the arrival curve $\alpha(t)$ is a wide-sense increasing function such that for all $s \leq t$ it holds that $R(t) - R(s) \leq \alpha(t-s)$, where $R(t) = \int_0^t r(x)\,\mathrm{d}x$ is the total amount of packets that has arrived to the server until time t. We refer the reader to [Le Boudec and Thiran, 2001] for a more thorough definition.

One common way to model arrival curves is with a burst-rate model, which means a quick burst of traffic, followed by a steady-state rate of incoming traffic. This can easily be expressed as

$$\alpha(t) = b + \rho \cdot t, \tag{1.2}$$

where b is the initial burst and ρ is the steady-state rate of the traffic, as illustrated by the purple line in Figure 1.9.

Service curves Similarly to how arrival curves may be used to provide an upper bound on the arriving traffic, we can use *service curves* to provide a lower bound on the available service provided by a server. A server's service curve β is also a wide-sense increasing function such that for any time-interval $[\tau, t]$ where the server has a nonzero queue, it must hold that $S(t) - S(\tau) \geq \beta(t-\tau)$. In words, it states that the output produced by the server during this interval is $S(t) - S(\tau)$ which is larger than the guaranteed service curve $\beta(t-\tau)$. One common service curve used to model a sever is a *rate-latency* service curve:

$$\beta(t) = [C \cdot (t-L)]^+, \tag{1.3}$$

where C is the processing rate of the server, and L is the latency, as illustrated by the yellow line in Figure 1.9.

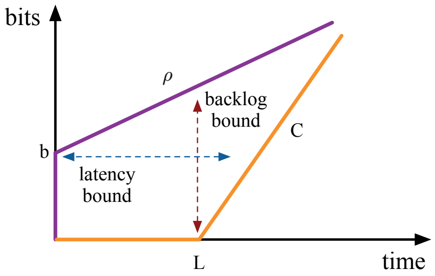


Figure 1.9 Illustration of how the arrival and service curves in network calculus can be used to derive an upper bound on the latency and backlog of a server. The arrival curve is illustrated in purple and the service curve in yellow.

Bounds on latency Together the service-curve and the arrival curve can be used to provide an upper bound on the backlog, i.e., the amount of traffic in the queue, and the latency of a node. For the case of a burst-rate arrival curve and a rate-latency service curve this is given by:

$$\text{latency bound} = L + \frac{b}{C}, \qquad \text{backlog bound} = b + \rho \cdot L. \tag{1.4}$$

Bounds on packet loss One can also use the service and arrival curves to derive deterministic bounds on the latency required for passing through a server, or the packet loss a server with a fixed queue-size might see. In [Le Boudec and Thiran, 2001], Le-Boudec derives an upper bound for the fraction of incoming packets that might be dropped if a server with a maximum queue size q^{max} has a service curve β and it sees traffic bounded by an arrival curve α. This upper bound is given by:

$$\hat{l} = \left[1 - \inf_{0<\tau} \frac{\beta(\tau) + q^{\text{max}}}{\alpha(\tau)}\right]^{+}. \tag{1.5}$$

The intuition behind this is that the larger the queue-size, or the more service is being provided by the server, the less packets will be dropped, given a specific arrival curve.

1.4 Related works

Despite the fact that the addressed problem in this thesis comes from very recent technology advancements (e.g. cloud computing, 5G, and virtual network functions), it is possible to abstract it in a way where related results can be found over quite a vast spectrum of older contributions. In an abstract way, the problem presented in this thesis can be decomposed into the following sub-components:

- providing end-to-end deadline guarantees for flows in a network,
- splitting end-to-end deadlines into local deadlines.

In the area of virtual network functions and data center management there are a number of works considering the problem of controlling virtual resources. However, many of them focus on orchestration, i.e. how the virtual resources should be mapped onto the physical hardware. A few works differ, however, in the way that they instead consider the problem of controlling the NFV graphs with respect to some end-to-end goals. For instance Lin et al. do a static one-time orchestration of an amount of resources sufficient to satisfy some end-to-end requests [Lin et al., 2014]. Sparrow presents an approach for scheduling a large number of parallel jobs with short deadlines [Ousterhout et al., 2013]. Cohen and similarly Shen address the issue of placement of VNFs within a physical network [Cohen et al., 2015; Shen et al., 2014]. A number of other works focusing on orchestration of VNFs, or scheduling of packet flows, can be found in [Moens and De Turck, 2014; Mehraghdam et al., 2014; Li and Qian, 2015], however, they do not address the issue of elastically scaling the capacity of the VNFs, nor do they allow for end-to-end constraints over the forwarding graphs.

Despite the dynamic nature of the traffic that the NFV graphs will encounter, there is only a few works that consider it and aim at designing an elastic and dynamic resource controller to counter the problem. Kuo, Mao et al., and Wang et al. address this problem by developing a mechanism for auto-scaling VNF resources to meet a user-specified performance goal, however they do not consider the issue of end-to-end constraints on the latency [Kuo et al., 2016; Mao et al., 2010; Wang et al., 2016]. Another work that addresses the problem of meeting performance goals despite dynamic traffic is [Leivadeas et al., 2016], where they do load-balancing with an SDN controller between the VNFs. Another work also combining flow scheduling and resource allocation is [Feng et al., 2016] where they develop a neat mathematical model used as foundation for their synthesis. Other works focusing on developing models of VNFs are [Faraci et al., 2017; Ren et al., 2016]. One work that do consider the constraint of an end-to-end latency requirement is [Li et al., 2016], where Li et al. present the design and implementation of NFV-RT that aims at controlling NFVs with soft Real-Time guarantees, allowing packets to have soft end-to-end deadlines.

A considerable amount of previous works address the deadline guarantee of a sequence of jobs that needs to be processed at a given node of a network [Tindell et al., 1994; Tindell and Clark, 1994; Pellizzoni and Lipari, 2007]. Gerber et al. [Gerber et al., 1995] proposed an alternate method to translate end-to-end deadlines over a directed graph of nodes into constraints on the activation periods of the tasks running at the intermediate nodes.

In the context of compositional analysis, previous works have addressed the problem of isolating and composing a single flow over a network of nodes. Lorente et al. [Lorente et al., 2006] extended the holistic analysis to the case with nodes running at a fraction of computing capacity (abstracted by a bounded-delay time partition with bandwidth and delay). Jayachandran and Abdelzaher [Jayachandran and Abdelzaher, 2008] developed several transformations ("delay composition algebra") to reduce the analysis of a distributed system to the single processor case. Serreli et al. [Serreli et al., 2009] proposed a component interface for chains of tasks activated sporadically and an intermediate deadline assignment, which minimizes the requested computing capacity. Similarly, Ashjaei et al. [Ashjaei et al., 2016] proposed resource reservation over each node along the path.

In the context of computation happening at "small" scale, it is worth mentioning the modular analysis by Hamann, Jersak, Richter, Ernst [Hamann et al., 2006]. Such a modular analysis, which found an application in the automotive domain, may well be a source of inspiration to analyze the schedulability within each node and the interaction between nodes. It is, however, orthogonal to our method which focuses on the policies to allow new flows of packets ("event streams" in the terminology of [Hamann et al., 2006]) to be admitted at run time.

There has been many works outside the area of network function virtualization that have addressed the enforcement of an end-to-end deadline of a sequence of jobs that is to be executed through a sequence of computing elements. In the holistic analysis [Palencia and Harbour, 2003] the schedulability analysis is performed locally. At global level the local response times are transformed into jitter or offset constraints for the subsequent tasks.

The idea of breaking end-to-end deadlines into local deadlines has also been exploited by several authors. Di Natale and Stankovic [Di Natale and Stankovic, 1994] proposed to split the end-to-end deadline proportionally to the local computation time or to divide equally the slack time. Marinca et al. [Marinca et al., 2004] proposed two methods to assign local deadlines ("Fair Laxity Distribution" and "Unfair Laxity Distribution") to balance the distribution of the slack among the flows. Later, Jiang [Jiang, 2006] used time slices to decouple the schedulability analysis of each node, reducing the complexity of the analysis. More recently, Hong et al. [Hong et al., 2015] formulated the local deadline assignment problem as a Mixed-Integer Linear Program (MILP) with the goal of maximizing the slack time. After local deadlines are assigned, the processor demand criterion was used to analyze distributed real-time pipelines [Rahni et al., 2008]. The number of local deadlines, however, is very high and makes the resulting optimization problem hard to solve. Jacob et al. [Jacob et al., 2016] proposed to split among local deadlines by using a *deadline ratio* $\rho \in (0,1)$ configuration parameter chosen at design-time.

In all the mentioned works, jobs have non-negligible execution times. Hence, their delay is caused by the preemption experienced at each function. In our context, which is scheduling of virtual network services, jobs are executed non-preemptively and in FIFO order. Hence, the impact of the local computation onto the E2E delay of a request is minor compared to the queueing delay. This type of delay is intensively investigated in the networking community in the broad area *queuing systems* [Kleinrock, 1975]. In this area, Henriksson et al. [Henriksson et al., 2004] proposed a feedforward/feedback controller to adjust the processing speed to match a given delay target.

Most of the works in queuing theory assumes a stochastic (usually Markovian) model of job arrivals and service times. A solid contribution to the theory of deterministic queuing systems is due to Baccelli et al. [Baccelli et al., 1992], Cruz [Cruz, 1991a; Cruz, 1991b], and Parekh & Gallager [Parekh and Gallager, 1993]. These results built the foundation for the *network calculus* [Le Boudec and Thiran, 2001], later applied to real-time systems in the *real-time calculus* [Chakraborty and Thiele, 2005]. The advantage of network/real-time calculus is that, together with an analysis of the E2E delays, the sizes of the queues are also modeled As in the cloud computing scenario the impact of the queue is very relevant since that is part of the resource usage which we aim to minimize, hence we follow this type of modeling.

2

Outline & Contributions

The goal of this thesis is to control a network of cloud nodes such that packet flows passing nodes within this network are guaranteed to meet their end-to-end deadlines. We wish to allow this network to be dynamic over time, meaning that nodes and flows should be allowed to join and leave the network over time, yet always guarantee that the end-to-end deadlines of the flows are met.

As hinted by the title of this thesis the process of solving this problem has been a journey. There has been a few sidetracks, but mainly it has been a steady progression and generalization towards the main goal. The structure of this thesis has been chosen to reflect this, with the added benefit of, hopefully, guiding the reader through this journey in a gentle and progressive way. The technical content of this thesis, presented over 6 chapters, can therefore be summarized as:

- A single cloud node;
  - In Chapter 3 we present AutoSAC combining admission control and horizontal scaling of VMs to allow dynamic traffic flowing to be processed within a deadline.
  - In Chapter 4 we combine horizontal and vertical scaling of VMs in order to control the processing capacity.
- A chain of cloud nodes;
  - In Chapter 5 we use horizontal scaling to ensure that a constant packet flow going through a chain of nodes meets an end-to-end deadline.
  - In Chapter 6 we generalize AutoSAC, and thereby allow for dynamic traffic to flow through the chain and meet an end-to-end deadline.
- A network of cloud nodes.
  - In Chapter 7 we generalize the AutoSAC again, and allow for multiple dynamic packet flows, flowing through a network of nodes.
  - Finally, in Chapter 8 we propose a way to manage a network of nodes and flows allowing for a dynamic network topology.

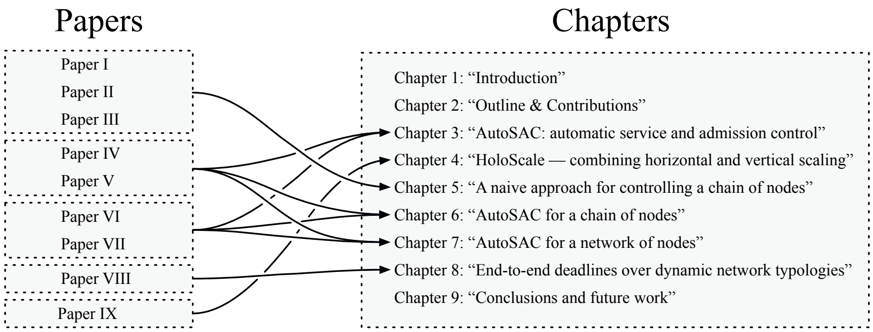

Papers
Chapters
Paper I
Paper II
Paper III
Paper IV
Paper V
Paper VI
Paper VII
Paper VIII
Paper IX
Chapter 1: "Introduction"
Chapter 2: "Outline & Contributions"
Chapter 3: "AutoSAC: automatic service and admission control"
Chapter 4: "HoloScale — combining horizontal and vertical scaling"
Chapter 5: "A naive approach for controlling a chain of nodes"
Chapter 6: "AutoSAC for a chain of nodes"
Chapter 7: "AutoSAC for a network of nodes"
Chapter 8: "End-to-end deadlines over dynamic network typologies"
Chapter 9: "Conclusions and future work"

Figure 2.1 Illustration of how the papers contribute to the different chapters.

Chapter 3: "AutoSAC: automatic service and admission control" In this chapter we present a way to combine automatic service and admission control (AutoSAC) for a node which could be a virtual network function, or a cloud application. As illustrated in Figure 2.2, we combine admission control with horizontal scaling of virtual machines in order to provide a deadline guarantee for the traffic flowing through the node. This traffic is also assumed to be dynamic and change over time.

The role of service-controller is to automatically scale the number of virtual machines in the node in order to adjust its processing capacity. The goal is to ensure that the node has sufficient processing capacity to process the incoming traffic. The role of the admission-controller is to ensure that the response-time of the node remains below a specific deadline.

Papers: IV, V,VI, and VII.

The admission and service controller presented in this chapter comes from papers VI–VII while the original problem formulation comes from papers IV and V.

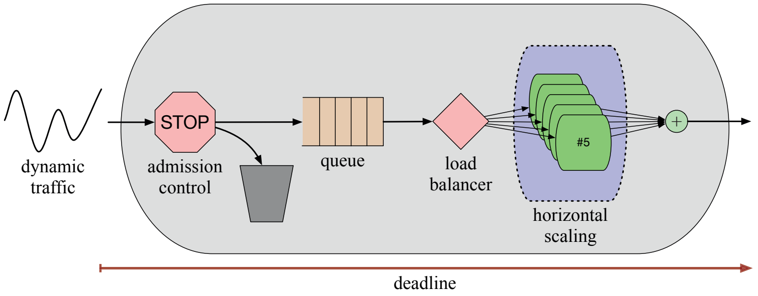


Figure 2.2 Illustration of AutoSAC presented in Chapter 3.

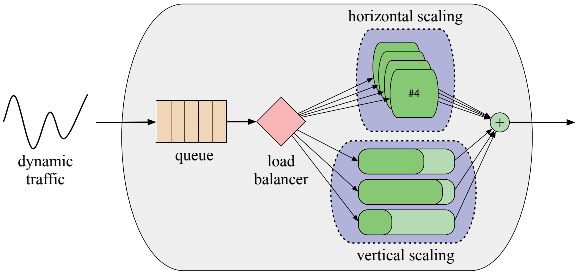


Figure 2.3 Illustration of HoloScale presented in Chapter 4.

Chapter 4: "HoloScale – combining horizontal and vertical scaling" In this chapter we travel along a tangent of the previous chapter. Here we pose the question of how cloud scaling can be improved if we could combine horizontal and vertical scaling of virtual machines, as illustrated in Figure 2.3. Recall that horizontal scaling is adding more/less of the same and vertical scaling is making the virtual machines larger/smaller. The main contribution is a way to combine the two scaling-methods allowing a smoother and quicker control of the processing capacity of the node. Note that we do not consider any deadlines within this chapter.

Papers: IX.

Chapter 5: "A naive approach to controlling a chain of nodes" In this chapter we introduce a naive approach for controlling a chain of nodes in order to ensure that the traffic flowing through the chain meets a specific end-to-end deadline. As illustrated in Figure 2.4 each node can horizontally scale the number of virtual machines in order to adjust its processing capacity. To simplify this problem, this chapter only considers a constant traffic rate for the packet flow. The main contributions include deriving an optimal schedule for the horizontal scaling along with a feedback-law allowing the nodes to handle stochastic variations to the incoming traffic.

Papers: I, II, and III.

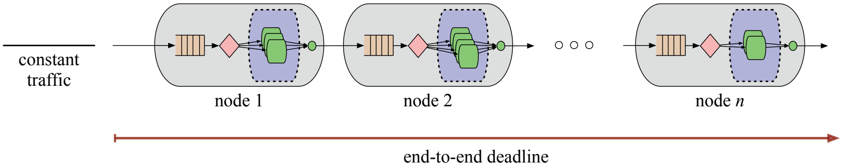


Figure 2.4 Illustration of the system presented in Chapter 5.

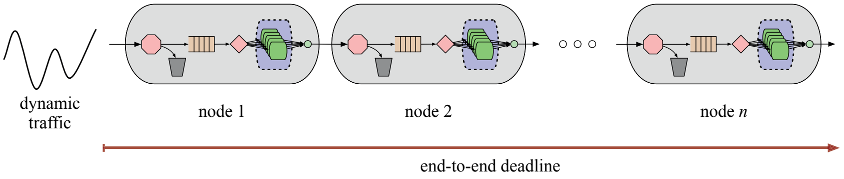


Figure 2.5 Illustration of AutoSAC for a chain, presented in Chapter 6.

Chapter 6: "AutoSAC for a chain of nodes" In this chapter we generalize AutoSAC presented in Chapter 3 to control a chain of nodes and allow for dynamic traffic to flow through the nodes, yet still provide end-to-end deadline guarantees. Along with this we also model uncertainties in the processing capacities of virtual machines. The main additions to AutoSAC include adding feedforward between the nodes, making the system more responsive. An overview of this system is shown in Figure 2.5.

Papers: IV, V,VI, and VII.

The main part of this chapter comes from papers IV and V. However, it should be noted that the admission controller and the node deadline assignment problem have been adjusted to be consistent with papers VI and VII. It should also be noted that the simulations have been updated to account for these changes.

Chapter 7: "AutoSAC for a network of nodes" On the journey towards allowing for a dynamic network topology, one of the lasts stops is to control a static network of nodes and flows. In this chapter we extend the AutoSAC of Chapter 6 to also work for a network of nodes with multiple flows passing through it. Each flow is allowed to have its own end-to-end deadline as well as dynamic traffic passing through it, as illustrated in Figure 2.6.

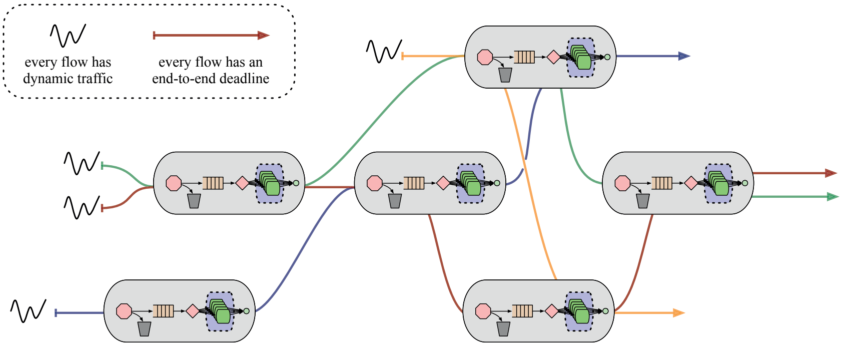

every flow has dynamic traffic
every flow has an end-to-end deadline

Figure 2.6 Illustration of AutoSAC for a network, presented in Chapter 7.

One of the contributions in this chapter is a way to differentiate between "internal" and "external" traffic as a way to improve the feedforward control. Another contribution, is a generalized version of the node-deadline assignment problem proposed in the previous chapter.

Papers: IV, V,VI, and VII.

The main part of this chapter originates from papers VI and VII. However, it should be noted that the convex utility function originates from papers IV and V.

Chapter 8: "End-to-end deadlines over dynamic network topologies" In this chapter we take a step back, and consider a high-level management of the nodes and flows in the network. We present a set of protocols controlling how flows and nodes are allowed to join and leave the network as well as a protocol for how the node-deadlines of the nodes in the network should be controlled over time. With these protocols we are able to formally prove that the end-to-end deadline of every flow in the network will be guaranteed. Moreover, these protocols allow for this to also hold while the network transitions between different network topologies, as illustrated in Figure 2.7. The theory presented in this chapter allows for both AutoSAC-controlled nodes in the network, as well as other control-strategies within the nodes.

Papers: VIII.

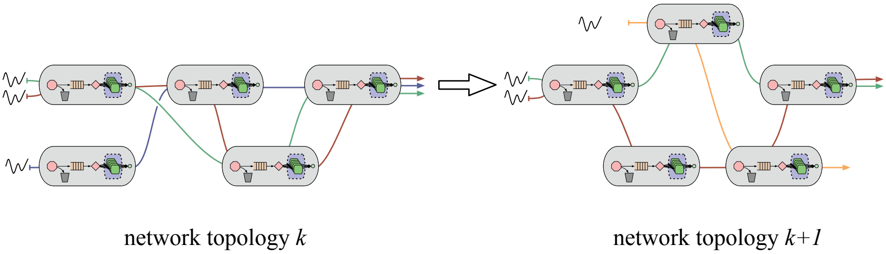


Figure 2.7 Illustration of a dynamic network topology which the methodology presented in Chapter 8 allows for.

Chapter 9: "Conclusions and future work" In this chapter we provide a summary and a discussion of the technical chapters in this thesis. Along with this we will also provide some interesting directions for future work. One interesting direction is to go towards a "dynamic network calculus". This involves modeling traffic flows using the bounds commonly found within network calculus, but to also combine them in a dynamic way. A quite promising direction would be to use this for dynamic priority-assignment of flows within a node.

2.1 Publications

In this section I list the publications used in this thesis, as well as my personal contributions for each of them.

Paper I

Millnert, V., E. Bini, and J. Eker (2016). "Cost Minimization of Network Services with Buffer and End-to-End Deadline Constraints". In: *REACTION, a Satellite Workshop at IEEE RTSS*. IEEE. Porto, Portugal, pp. 31–37.

This paper was also republished in the ACM SIGBED Review:

Millnert, V., E. Bini, and J. Eker (2018). "Cost minimization of network services with buffer and end-to-end deadline constraints". *ACM SIGBED Review,* ACM, pp. 39–45.

These papers treat the problem of controlling a chain of connected cloud function through which a constant stream of packets flow. The packets flowing through the chain must be processed by each of the cloud function before a specific end-to-end deadline.

Each cloud function consists of a number of virtual machines, and the goal of this paper is to study how many virtual machines each cloud functions should have at any point in time. In essence, we wish to derive an offline schedule for the virtual machines in the different cloud functions. By following this schedule the packets are guaranteed to make it through the network on time. Moreover, the derived schedule ensures that the operational costs for hosting the VMs are minimized.

Authors contribution. Apart from deriving the model for the chain of cloud services and stating the problem formulation, V. Millnert was the lead author in writing the manuscript as well as the lead developer of the solution. Naturally, this was done with the feedback and comments from both J. Eker and E. Bini.

Paper II

Millnert, V., J. Eker, and E. Bini (2017). "Feedback for increased robustness of forwarding graphs in the cloud". *Journal of Systems Architecture,* Elsevier, pp. 68–76.

This paper builds on Paper I and extends it by adding a feedback-law. With this addition it becomes possible to allow the chain of cloud functions to also handle a stochastic arrival rate as well as disturbances.

Without the new feedback law, the cloud functions would eventually drift away from the desired schedule, but with the addition of the feedback-law it is possible to guarantee that the cloud functions will return back to the desired schedule, even in the presence of disturbances.

Authors contribution. V. Millnert was the lead author in writing the extended journal manuscript as well as responsible for developing the new feedback law. He was also responsible for building and evaluating the simulations. J. Eker and E. Bini both helped with comments and feedback on the manuscript as well as the theoretical work.

Paper III

Millnert, V., J. Eker, and E. Bini (2016). "Cost Minimization of Network Services with Buffer and End-to-End Deadline Constraints". *Lund University, Technical Reports TFRT-7648*.

This paper is the technical report in which we collected things that did not fit into Paper I and II. Among other things, it contains the proofs and some additional analysis.

Authors contribution. See the contributions for Papers I and II, since this is the technical report which contains the material used for both of those papers.

Paper IV

Millnert, V., J. Eker, and E. Bini (2017). "Dynamic Control of NFV Forwarding Graphs with End-to-End Deadline Constraints". In: *IEEE International Conference on Communications*. **Best Paper Award**. IEEE. Paris, France, pp. 1–7.

In this work we generalized the model of Papers I–III to also include uncertainties and dynamic behavior. The uncertainties included things such as available processing capacity of virtual machines or how long it would take to start a new virtual machine. The dynamic behavior comes from allowing a highly varying traffic (the behavior of which is based on real traffic data gathered from the Swedish University Network).

Authors contribution. V. Millnert was the lead author and investigator in this work. He developed the new model and derived the solutions (i.e., the admission controller and the service controller). He also implemented and evaluated the simulations. J. Eker and E. Bini assisted with both feedback and comments throughout the process of this work.

Paper V

Millnert, V., E. Bini, and J. Eker (2017). "AutoSAC: Automatic Scaling and Admission Control of Forwarding Graphs". *Annals of Telecommunications,* Springer, pp. 15–12.

This work is an extension of Paper V, in which we included more analysis and evaluation, as well as a more general control-law for scaling the virtual machines in a cloud function.

Authors contribution. V. Millnert was responsible for extending Paper IV into a journal version. Hence, he was responsible for developing the extra material (i.e., generalized control-law) and the added analysis. He was also the lead author.

Paper VI

Millnert, V., J. Eker, and E. Bini (2018). "Achieving Predictable and Low End-to-End Latency for a Network of Smart Services". In: *IEEE Global Communications Conference*. IEEE. IEEE, Abu Dhabi, UAE, pp. 1–7.

This work builds on Papers IV and V with the addition being a generalization towards a network of cloud functions. With this, we allow for multiple flows to pass through the network of cloud functions. Along with this we introduced a concept of differentiating between "internal" and "external" traffic, which further improved how the feedforward-information was passed between the cloud functions in the network.

Other contributions in this work included improvements to the admission controller first proposed in Papers IV and V. With the new admission controller it becomes possible to perform the check of whether new packets should be admitted or not using a simple if-statement, making it simple to implement in a real system.

Authors contribution. V. Millnert was mainly responsible for the generalization of Papers IV and V into this work. He developed the network system model, the new control-laws, and the new feedforwad scheme. He was also responsible for developing the new simulation and evaluation. V. Millnert was the lead author and wrote the manuscript with feedback and comments from both J. Eker and E. Bini.

Paper VII

Millnert, V., J. Eker, and E. Bini (2018). "Achieving Predictable and Low End-to-End Latency for a Cloud-Robotics Network". *Lund University, Technical Reports* TFRT-7655.

This paper is a technical report in which we published, internally, theory and analysis that did not fit into Paper VI due to lack of space. Among other things, this includes more evaluation as well as a proposed way to estimate how the routing of the traffic between different cloud functions in the network changes over time.

Authors contribution. See the contributions for Paper VI.

Paper VIII

Millnert, V., J. Eker, and E. Bini (2019). “End-to-End Deadlines over Dynamic Topologies”. In: *Euromicro Conference on Real-Time Systems*. Stuttgart, Germany, pp. 1–22.

In this work we take a step back from what was done in Papers IV–VII and instead assume that every cloud function in the network is able to guarantee that it will meet its local node deadline (as is done in Papers IV–VII). With this assumption in mind, the focus of this paper is on controlling the local deadlines of the cloud services. The goal is to do this in a dynamic way so that flows and nodes can be allowed to join and leave the network.

The contributions in this paper are a main theorem which propose a set of sufficient conditions under which we can guarantee that all the end-to-end deadlines of all the flows will be met. Moreover, we propose a set of protocols which dictates how flows and nodes should be join/leave the network. We prove that by following these protocols, there will be no violation to the main theorem.

Authors contribution. V. Millnert was the lead author and developer. He developed the problem formulation and derived the solution together with E. Bini. He was also responsible for developing the simulations and evaluations. V. Millnert was the lead author of the manuscript assisted by E. Bini and J. Eker.

Paper IX

Millnert, V. and J. Eker (2019). “HoloScale: Combining Horizontal and Vertical Scaling”. In preparation.

In this work we propose a way to combine horizontal and vertical scaling of cloud resources in order to allow for a very smooth and fast control of the available computational power of a cloud function. Finally, we also provide a way to derive a region of safe control parameters given some characteristics of the cloud function.

Authors contribution. In this work V. Millnert is solely responsible for all work.

3

AutoSAC: automatic scaling and admission control

In this chapter we present an idea of how to control a virtual network function, or a cloud service, in a way that ensures that traffic flowing through it is guaranteed to meet a specific deadline. As mentioned in Chapter 2, this is the first step towards the final goal of guaranteeing end-to-end deadlines for flows passing through a dynamic network of nodes, where by dynamic network we mean that the topology of the network is allowed to change over time.

The intuition behind the ideas presented in this chapter is to combine admission control, i.e., deciding whether new traffic should be admitted into the node or not, with horizontal scaling of virtual machines, or containers, as illustrated in Figure 3.1. The goal is to do this in a way such that the response time of the node, or the time it takes an admitted packet to be processed by the node, is below a deadline.

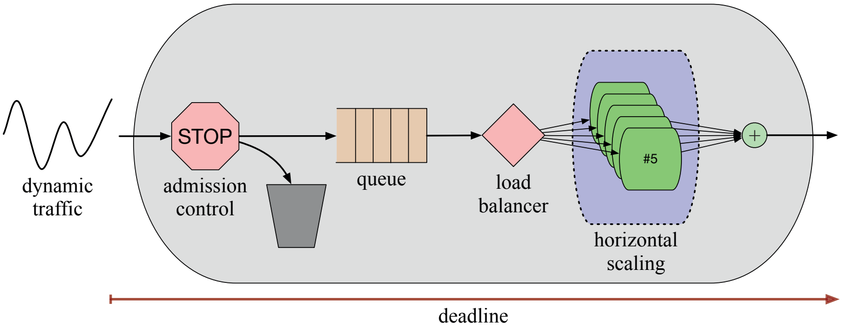


Figure 3.1 Illustration of the cloud service (or virtual network function) considered in this chapter. The goal is to control the traffic being admitted into the node, as well as to control how many virtual machines it should have active at any given point in time. The goal is to do this in a way that guarantees that the packets flowing through the node are processed within a deadline $\mathscr{D}$.

3.1 Modeling of a cloud node

Before presenting how we will combine the admission control and the horizontal scaling, we will present a mathematical model of the system. As illustrated in Figure 3.1 a node we consider is modeled with some key entities: *incoming traffic*, *admission controller*, *a queue*, *a load balancer*, *a number of running execution environments (typically virtual machines or containers)*, and finally *outgoing traffic*.

In this thesis we use a *fluid approximation* of the traffic, meaning that the traffic flowing as the nodes, and later the networks of nodes, are modeled through infinitesimally small packets. One reason for this is because of the simplified analytical expressions, and another one is that in a benchmarking study it was shown that a typical virtual machine can process around 0.1–2.8 million packets per second, [Bonafiglia et al., 2015]. Hence, in this work the number of packets flowing through the functions is assumed to be in the order of millions of packets per second, supporting the choice of a fluid model.

Traffic flow At time t the node will see traffic arriving to the node at an *incoming traffic rate* of $r(t) \in \mathbb{R}^+$ packets per second. This traffic can either be admitted into the queue of the node, or be discarded. The rate by which traffic is admitted into the queue is given by the *admission rate* $a(t) \in [0, r(t)]$. It is the job of the *admission controller* to control this rate, and later in Section 3.2 we will present a policy for doing so. Finally the rate by which traffic is being processed by the node, and is leaving it with, is modeled by the *service rate* $s(t) \in \mathbb{R}^+$. From these signals we can also define the integrals

$$S(t) = \int_0^t s(x)\,\mathrm{d}x, \qquad A(t) = \int_0^t a(x)\,\mathrm{d}x, \qquad R(t) = \int_0^t r(x)\,\mathrm{d}x. \tag{3.1}$$

which allows us to model the amount of work in the queue as

$$q(t) = A(t) - S(t), \quad q(t) \in \mathbb{R}^+. \tag{3.2}$$

Here is should be noted that this is assumed to model all the packets currently inside the node. Hence, there is no distinction between packets being in the queue, or being within the execution environment (or the virtual machines). This is an artifact from the fluid approximation, and a useful analogy is that all packets are modeled as being in the queue when they are in the node. The virtual machines then act as a “throttle”, controlling the rate by which packets leave this queue. The more virtual machines running in the node, the more processing capacity is available and the larger this “throttle” is.

Machine model At any time instance, the node will have $m(t) \in \mathbb{Z}^+$ *virtual machine instances* up and running. Each instance (or replica) is assumed to be capable of processing packets and corresponds to a virtual machine, a container, or a process running in the OS. It is possible to control the number of running instances by sending a *reference signal* $m^{\mathsf{ref}}(t) \in \mathbb{Z}^+$ to the service controller. However, as explained in Chapter 1, it takes some time to start/stop instances since an instantiation of the service is always needed. We denote this as the *time overhead* Δ. This time-overhead is for simplicity considered to be symmetrical (meaning it is assumed to take the same amount of time to start a new instance as to kill one). In real life it is usually quicker to kill one than to boot one up. Moreover, it is assumed that this time-overhead is constant over time. Naturally, this is not always the case in a real system, but any variations in Δ can be modeled through the processing uncertainty presented later, in Chapter 6. With this time-delay Δ, the number of instances running at time t is given by

$$m(t) = m^{\mathsf{ref}}(t - \Delta).$$

The machine instances in the node are assumed to be homogenous, meaning that they are replicas of the same virtual machine image, or container. Therefore, by modeling the *nominal service rate* of a single machine instance by $\bar{s}$ packets per second, we can model the *maximum processing capacity* $s^{\text{cap}}(t)$ of the node as

$$s^{\text{cap}}(t) = m(t) \cdot \bar{s}. \tag{3.3}$$

Throughout the thesis we will interchangeably the *available processing capacity* and *maximum processing capacity* to refer to $s^{\text{cap}}(t)$.

Processing of packets The packets in the queue are stored and processed in a FIFO manner. As mentioned earlier, the processing capacity of the node can be seen as a throttle, adjusting at what rate traffic is leaving the node. This rate is given by the service rate, $s(t)$, which we can now define as

$$s(t) = \begin{cases} s^{\text{cap}}(t) & \text{if } q(t) > 0, \\ \min\left\{a(t),\ s^{\text{cap}}(t)\right\} & \text{else.} \end{cases} \tag{3.4}$$

Here one can again see how the maximum processing capacity acts as a “throttle”. One can also see that the throttle can only ensure that traffic leave the node at a rate of $s(t)$ if there are packets in the queue, or if the admission rate of the node is larger than this. If the queue is empty, and the admission rate is lower than the maximum processing capacity, then the rate $s(t)$ is limited by this, leading to $s(t) = a(t)$.

Deadline and latency The goal of this chapter is to ensure that an admitted packet never spend a time longer than $\mathscr{D}$ time-units within the node. Therefore, if a packet enters the node at time t, it should exit the node before time $t + \mathscr{D}$.

The time that a packet which exits the node at time t has spent inside the node is denoted the *response time* or *node latency* $L(t)$ of the node, and is given by:

$$L(t) = \inf\{\tau \geq 0 : A(t-\tau) \leq S(t)\}.$$

One should note here that this model of the response-time, or the latency, accounts for all possible delays within the node. This includes processing delays, internal communication latency, queueing delay, etc. This means that if the latency of the node is $L(t)$ a packet exiting the node at time t was admitted into the node at time $t - L(t)$.

The choice of modeling the latency of the node in this way, e.g., as queueing delay only, can be motivated from a study by Google, where they profiled the latency in a datacenter [Kapoor et al., 2012]. They showed that less than 1% ($\approx 1\ \mu s$) of the latency occurred was due to the propagation in the network fabric. The other 99% ($\approx 85\ \mu s$) occurred somewhere in the kernel, the switches, the memory, or the application. Since it is difficult to say exactly which of this 99% is due to processing, or queuing, we make the abstraction of considering queuing delay and processing delay together, simply as queuing delay.

If one would wish to account for a specific processing time of the packets, one could easily do so. For instance, should this processing time be constant, one could account for this by simply reducing the deadline $\mathscr{D}$ by this amount: $\tilde{\mathscr{D}} = \mathscr{D} - c$, where c is the processing time. This would in turn lead us back to the case used in this thesis, so for increased readability the processing time of packets is omitted from this work.

Along with the definition of the latency $L(t)$, it is also useful to define the *expected latency* of the node as $\bar{L}(t)$. This is commonly referred to as the "virtual delay" in the field of network calculus [Le Boudec and Thiran, 2001]. It means that it is the time that a packet entering the node at time t will need in order to pass through the node, and can be computed as:

$$\bar{L}(t) = \inf\left\{\tau \geq 0 : A(t) \leq S(t) + \int_t^{t+\tau} m(x) \cdot \bar{s}\ \mathrm{d}x\right\}. \tag{3.5}$$

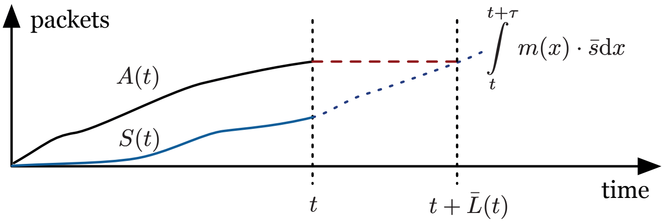


Figure 3.2 Illustration behind Equation (3.5). $\bar{L}(t)$ is the time τ required until $S(t) + \bar{s}\int_t^{t+\tau} m(x)\mathrm{d}x$ is greater then $A(t)$. In the illustration the integral is illustrated by the blue dashed line, $A(t)$ by the black line, and $S(t)$ by the solid blue line.

The intuition behind Equation (3.5) is illustrated in Figure 3.2, and is that we are looking for the minimum $\tau \geq 0$ such that $S(t+\tau) = A(t)$, or in other words such that at time $t+\tau$ the node will have processed all the packets that has entered the node up until time t. One should note that $\bar{L}(t)$ is non-causal, and computing it requires information of $m(t)$ for the future. Since $m(t) = m^{\text{ref}}(t-\Delta)$, this information does however exist for a time-period up until Δ time-units into the future. Therefore, as long as $\bar{L}(t) \leq \Delta$ one can compute it.

Problem formulation

The goal of this chapter is to derive a service-controller and an admission-controller that guarantee that packets that pass through the node meet their deadline $\mathscr{D}$. This should be done using as few virtual machines as possible while still achieving as high throughput as possible. In other words, one wish to discard as little traffic as possible, and use as few virtual machines as possible, yet still ensure that the response time is less than the deadline.

To evaluate this, we have derived a simple, yet intuitive *utility function* $u(t)$. Later in Section 3.2, the utility function is used to derive an automatic service and admission controller, denoted AutoSAC.

Utility function The goal in this chapter is to derive ways of controlling the number of active virtual machines in a node. Informally, this goal can be described as trying to ensure that *"the end-to-end deadlines of the different packet-flows are met, while using as few virtual machines as possible and while discarding as few packets as possible"*. As an aid when evaluating this goal we propose three formal metrics, defined in (3.6):

a) *availability* $u^{\text{a}}(t)$ – this is meant to capture the ratio of the incoming packets being admitted to the node. If there is a high ratio, then it means that most of the incoming traffic is being admitted into the node.

b) *efficiency* $u^{\text{e}}(t)$ – this is meant to capture what fraction of available processing capacity is being used. A high ratio means that most of the available processing capacity is being used to process traffic.

c) *utility* $u(t)$ – this is a combination of $u^{\text{a}}(t)$ and $u^{\text{e}}(t)$ meant to capture the overall performance of the node.

The intuition behind the choice of these metrics is that it is typically easy to have either a high efficiency or a high availability, but not both at the same time. One can for instance choose to overallocate the number of virtual machines in a node, resulting in a high availability but a poor efficiency (since it will lead to a poor utilization of the available processing capacity). Similarly, one can instead choose to have a high efficiency by not allocating enough virtual machines to the node, thus forcing the node to discard many packets. The *utility* $u(t)$ is defined as

$$u(t) = u^{\mathsf{a}}(t) \cdot u^{\mathsf{e}}(t), \tag{3.6}$$

with $u^{\mathsf{a}}(t)$ and $u^{\mathsf{e}}(t)$ given by

$$\begin{aligned} u^{\mathsf{a}}(t) &= \begin{cases} s(t)/r(t) & \text{if } L(t) \leq \mathscr{D} \\ 0 & \text{if } L(t) > \mathscr{D} \end{cases} \\ u^{\mathsf{e}}(t) &= \frac{s(t)}{s^{\text{cap}}(t)} \end{aligned} \tag{3.7}$$

One should notice that Equation (3.7) assumes that should a packet miss its deadline it is considered useless, which is why $u^{\mathsf{a}}(t)$ is evaluated as zero in such a case. It is thus favorable to use admission control to drop packets that have a high probability of missing their deadline in order to make room for subsequent packets.

3.2 Controller design

In this section an automatic service and admission-controller (AutoSAC) is derived. Figure 3.3 illustrates an overview of the different parts of AutoSAC and the information flow it uses. The service controller measures the incoming traffic and the amount of work in the queue in order to estimate how much processing capacity it will need in the near future. Due to the time-overhead in starting/stopping virtual machines, and thereby changing the processing capacity of the node, it might be difficult to predict and exactly match the necessary processing capacity. Therefore the admission controller might have to discard packets sometimes. However, in order for the node to not discard unnecessarily many packets, the admission controller use feedback from the control signal and the current queue size.

The difficulty when deriving AutoSAC lies in the different time-scales for starting/stopping virtual machines, the node deadline, as well as the rate-of-change of the input. They are all assumed to be of different orders of magnitudes, given by Table 3.1. However, these timing assumptions will be exploited later.

Property	timing assumption
Long-term trend change of the traffic intensity	1 min – 1 h
Time-overhead Δ for changing the number of VMs	1 s – 1 min
Deadline $\mathscr{D}$ for the traffic flowing through the node	1 ms – 1 s

Table 3.1 Timing assumptions for the deadline, the change-of-rate of the input, and the overhead for changing the service-rate.

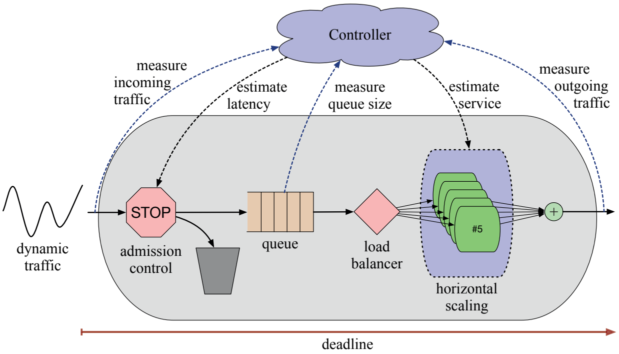


Figure 3.3 Overview of the automatic service and admission-controller highlighting that it uses feedback from the true performance of the nodes to estimate the time it will take incoming packets to pass through the node.

Admission controller

As mentioned earlier, the traffic flowing through the node has a *deadline* $\mathscr{D}$. In order to ensure that the traffic that is admitted into the node is guaranteed to meet the deadline, we will derive a simple admission controller. The basic idea is straight forward: compare the deadline $\mathscr{D}(t)$ with the *expected latency* $\bar{L}(t)$. If the expected latency, $\bar{L}(t)$, is larger than the deadline, $\mathscr{D}$, then the admission controller should discard the incoming traffic, giving us the following *admission control policy*:

$$\begin{cases} \text{admit traffic} & \text{if } \mathscr{D} > \bar{L}(t), \\ \text{discard traffic} & \text{if } \mathscr{D} \leq \bar{L}(t). \end{cases} \tag{3.8}$$

As we will show in Chapter 6, Theorem 6.1, one can ensure that $\mathscr{D} > \bar{L}(t)$ by ensuring that the following inequality hold:

$$A(t) < S(t) + S^{\text{cap}}(t+\mathscr{D}) - S^{\text{cap}}(t), \tag{3.9}$$

where $S^{\text{cap}}(t)$ is defined by:

$$S^{\text{cap}}(t) = \int_0^t m(x) \cdot \bar{s} \, \mathrm{d}x. \tag{3.10}$$

The reason for this follows from Equation (3.5), since

$$A(t) \leq S(t) + S^{\text{cap}}(t+\mathscr{D}) - S^{\text{cap}}(t) \quad \Rightarrow \quad \mathscr{D} \geq \bar{L}(t),$$

and because $\int_t^{t+\tau} m(x) \cdot \bar{s} \, \mathrm{d}x = S^{\text{cap}}(t+\tau) - S^{\text{cap}}(t)$ and because $A(t)$, $S(t)$, and $S^{\text{cap}}(t)$ are all non-decreasing. Therefore, to enforce the admission policy (3.8), it is sufficient to instead compute inequality (3.9). This is good because $\bar{L}(t)$ is typically expensive to compute as it usually involves at least one linear search, whereas computing (3.9) can be done in constant time (assuming that the integrals are readily available). Doing this instead, leads to the admission policy:

$$\begin{cases} \text{admit traffic} & \text{if } A(t) < S(t) + S^{\text{cap}}(t+\mathscr{D}) - S^{\text{cap}}(t), \\ \text{discard traffic} & \text{else.} \end{cases}$$

The remaining question is then how much traffic should be admitted or discarded? Naturally, whenever there is a strict inequality in (3.9), we should admit as much traffic as possible, i.e., $a(t) = r(t)$. When instead there is equality, some care must be taken and packets may have to be discarded in order to ensure that the inequality is not violated. In such a case, the rate by which the admission controller can admit traffic is governed by the rate by which $S^{\text{cap}}(t+\mathscr{D})$ is growing, i.e., by $s^{\text{cap}}(t+\mathscr{D})$. Therefore, the final admission policy will be given by:

$$a(t) = \begin{cases} r(t) & \text{if } A(t) < S(t) + S^{\text{cap}}(t+\mathscr{D}) - S^{\text{cap}}(t), \\ \min\big(r(t), s^{\text{cap}}(t+\mathscr{D})\big) & \text{else.} \end{cases} \tag{3.11}$$

The reason for the min()-statement is because it cannot admit packets at a rate higher than $r(t)$, should $r(t) < s^{\text{cap}}(t+\mathscr{D})$. In Figure 3.4 a block diagram of the admission controller is illustrated. It shows that by simple integration and time-delays, the admission policy can be computed continuously and in constant time.

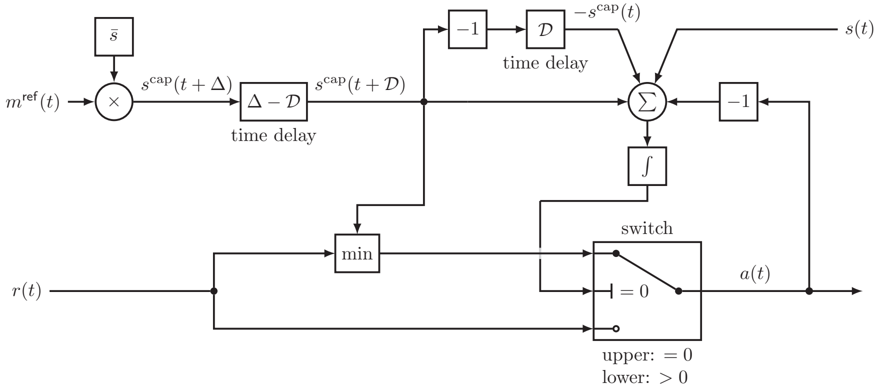


Figure 3.4 Block-diagram of the admission controller. It uses feedback from both admission rate $a(t)$, the service rate $s(t)$, as well as the control-signal $m^{\text{ref}}(t)$.

Service controller

The goal for the service-controller is to find $m^{\text{ref}}(t)$ such that the utility function is maximized once the reference signal is realized, i.e. such that $u(t+\Delta)$ is maximized.

We begin by expanding the utility function (3.6) giving us the following expression

$$u(t) = u^{\mathsf{a}}(t) \cdot u^{\mathsf{e}}(t) = \frac{s^2(t)}{s^{\text{cap}}(t) \cdot r(t)}.$$

Along with this, we note that the service rate $s(t)$ can be approximated to be either the current maximum capacity of the node, $s^{\text{cap}}(t)$, or the input rate $r(t)$

$$s(t) \approx \min\{s^{\text{cap}}(t),\ r(t)\}. \tag{3.12}$$

The intuition behind using the *min*-statement is that the node cannot process packets at a faster rate than what they are entering the node for a prolonged period of time. Likewise, it cannot process packets at a rate higher than the capacity of the node when the input were to be higher than this. We can thus approximate the utility function through:

$$u(t) \approx \begin{cases} \dfrac{\left(s^{\text{cap}}(t)\right)^2}{s^{\text{cap}}(t) \cdot r(t)} = \dfrac{s^{\text{cap}}(t)}{r(t)}, & \text{if } s^{\text{cap}}(t) \leq r(t), \\ \dfrac{r^2(t)}{s^{\text{cap}}(t) \cdot r(t)} = \dfrac{r(t)}{s^{\text{cap}}(t)}, & \text{else.} \end{cases}$$

With $s^{\text{cap}}(t)$ given by (3.3) the utility function can finally be approximated as

$$u(t) \approx \begin{cases} \dfrac{m(t) \cdot \bar{s}}{r(t)}, & \text{if } m(t) \cdot \bar{s} \leq r(t), \\ \dfrac{r(t)}{m(t) \cdot \bar{s}}, & \text{else.} \end{cases} \tag{3.13}$$

Since the goal is to find $m^{\text{ref}}(t)$ in order to maximize $u(t+\Delta)$ one needs knowledge of $r(t+\Delta)$ which is not available. However, one can estimate the future input-rate using a linear extrapolation of the (low-pass filtered) input rate:

$$\hat{r}(t) = r(t) + \Delta \cdot \frac{\mathrm{d}r(t)}{\mathrm{d}t}.$$

Formulating the control-law. With the approximations above, the control-law for $m^{\text{ref}}(t)$ can be posed as the following optimization problem:

$$m^{\text{ref}}(t) = \begin{cases} \arg\max\limits_{x \in \mathbb{Z}^+} \{x/\kappa(t)\}, & \text{if } x \leq \kappa(t), \\ \arg\max\limits_{x \in \mathbb{Z}^+} \{\kappa(t)/x\}, & \text{else,} \end{cases} \tag{3.14}$$

where $\kappa(t) \in \mathbb{R}^+$ is the real-valued number of instances needed to exactly match the predicted incoming rate:

$$\kappa(t) = \frac{\hat{r}(t)}{\bar{s}}. \tag{3.15}$$

In (3.14) one can see that the upper case is maximized when x is as large as possible, but since this case is only valid when $x \leq \kappa(t)$ it leads to $x = \lfloor \kappa(t) \rfloor$. Similarly, the lower case is maximized when x is as small as possible, but since this case is valid for $x \geq \kappa(t)$ it leads to $x = \lceil \kappa(t) \rceil$, leading to the final control-law:

$$m^{\text{ref}}(t) = \begin{cases} \lfloor \kappa(t) \rfloor, & \text{if } \lfloor \kappa(t) \rfloor \lceil \kappa(t) \rceil \geq \kappa^2(t), \\ \\ \lceil \kappa(t) \rceil, & \text{else.} \end{cases} \tag{3.16}$$

Approximation of the control-law Sometimes there is a need to have a large number of virtual machines running in the nodes, and in such a case this control-law may be simplified. In Theorem 3.1 we show that for such a case the control-law can be simplified into $m_i^{\text{ref}}(t) = \lfloor \kappa_i(t) \rceil$.

THEOREM 3.1
When the node is having a large number of virtual machines running, $m_i^{\text{ref}}(t)$ can be approximated to the nearest integer and computed as $m_i^{\text{ref}}(t) = \lfloor \kappa_i(t) \rceil$. □

Proof Substituting $m_i^{\text{ref}}(t)$ for y and $\kappa_i(t)$ for x we can start from

$$y = \begin{cases} \lfloor x \rfloor, & \text{if } \lfloor x \rfloor \lceil x \rceil \geq x^2 \\ \lceil x \rceil & \text{else} \end{cases} \tag{3.17}$$

allowing us to write x as $x = \lfloor x \rfloor + \rho$, with $\rho \in [0, 1)$. If $\rho = 0$ then $\lfloor x \rfloor = \lceil x \rceil = x$, implying that $y = x$. Otherwise, it follows that

$$\begin{gathered} \lfloor x \rfloor \lceil x \rceil \geq x^2 \\ \lfloor x \rfloor \cdot (\lfloor x \rfloor + 1) \geq (\lfloor x \rfloor + \rho)^2 \\ \lfloor x \rfloor^2 + \lfloor x \rfloor \geq \lfloor x \rfloor^2 + 2\rho \lfloor x \rfloor + \rho^2 \\ \lfloor x \rfloor \geq 2\rho \lfloor x \rfloor + \rho^2 \\ \rho \leq \frac{1}{2} - \frac{\rho^2}{2 \lfloor x \rfloor}. \end{gathered}$$

Hence, Equation (3.17) can be rewritten as

$$y = \begin{cases} \lfloor x \rfloor, & \text{if } \rho \leq \frac{1}{2} - \frac{\rho^2}{2\lfloor x \rfloor} \\ \lceil x \rceil & \text{else} \end{cases}$$

which for large x then becomes

$$y = \lfloor x \rceil.$$ □

Comparison between control-law and the approximation To give some intuition about the difference of using the control-law (3.16) for computing $m^{\text{ref}}(t)$ and using the approximation proposed in Theorem 3.1, we will show some evaluation and analysis here. First, in Figure 3.5 the difference of m^{ref} is illustrated for different values of κ. There, one can see that when κ grows larger than 4-5 the difference between the two methods becomes very small.

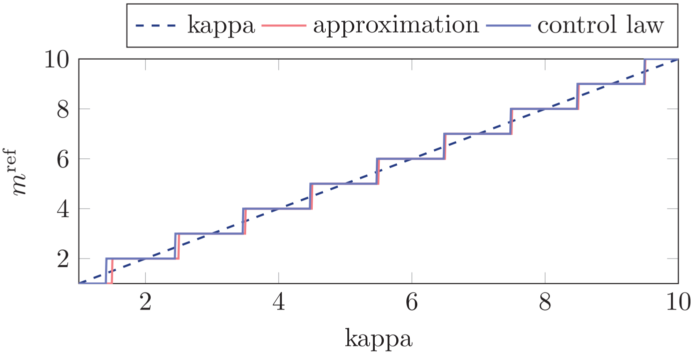
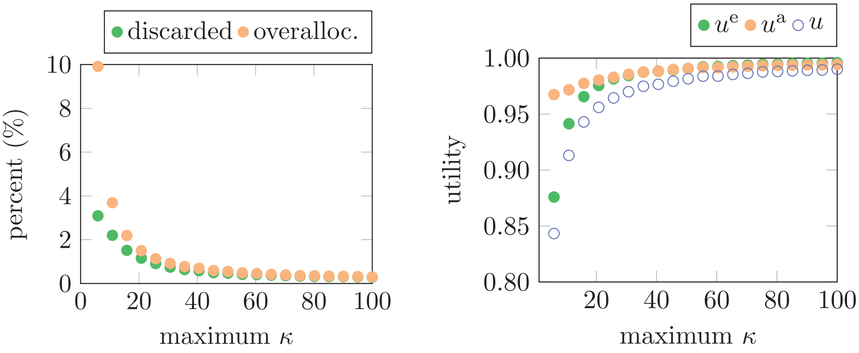


Figure 3.5 Figure illustrating the difference between m^{ref} when using the control-law (3.16) and the approximation, presented in Theorem 3.1. One can see when $\kappa > 4$ there is almost no difference at all between the two methods.

In Figure 3.6 we illustrate how different values of κ affect the performance of the node. This evaluation is made using the simulation methodology presented later, in Section 3.3, but where we controlled the nominal processing capacity of the virtual machine instance $\bar{s}$ such that the simulations would have a maximum κ during the simulations. More specifically, $\bar{s}$ was chosen such that $\bar{s} = \max_t \big(r(t)\big)/\kappa$, where $\kappa = \{1, 2, \ldots, 100\}$. As described in Section 3.3, every simulation had a peak traffic rate of $\max_t \big(r(t)\big) = 10\,000\,000$. This means that in Figure 3.6, maximum κ, means that during the simulations to match the peak traffic, the node had to have κ machines on.

The results of Figure 3.6 show that the performance of the system reach a plateau for $\kappa > 30$. This is shown in Figure 3.6(a) where one can see how the percentage of the traffic being discarded is affected as well as how the amount of overallocation is also affected. In Figure 3.6(b) one can see how the number of instances used to match peak capacity affect efficiency, the availability, and the utility of the node. Again, one can see that for $\kappa > 30$ the performance is reaching a plateau.

The main reason for the increase in performance when κ grows above $\kappa > 30$ is because it allows for a finer granular control of the number of virtual machines allocated to the node. It will become easier for the node to precisely match the required processing capacity.



(a) Evaluation of how κ affect the amount of overallocation in the node, as well as the amount of traffic being discarded by admission control.

(b) Evaluation of how κ affect the efficiency $u^e(t)$, the availability $u^a(t)$, and the utility $u(t)$ of the node. Naturally, this correlates to Figure 3.6(a).

Figure 3.6 Figure illustrating how κ affects the performance of the system. The x-axis shows the maximum κ obtained during the simulation. The peak traffic rate during the simulations was $\max\big(r(t)\big) = 10\,000\,000$, so in order to vary $\max_t\big(\kappa(t)\big)$ the nominal service-rate $\bar{s}$ was between the simulations.

3.3 Evaluation

In this section, the automatic service and admission-controller (AutoSAC) developed in Section 3.2 is evaluated. First, we will illustrate how we simulated the cloud node with randomly generated parameters, and then show AutoSAC is compared with two other "industry" methods for scaling cloud services. The comparison is done using a Monte Carlo simulation where the parameters of a node were randomly generated and then simulated, using a real traffic trace as the basis for the incoming traffic intensity. The traffic trace used as input was gathered over 120 hours from a port in the Swedish University NETwork (SUNET) and then scaled to have a peak of 10 000 000 packets per second as shown in Figure 3.7.

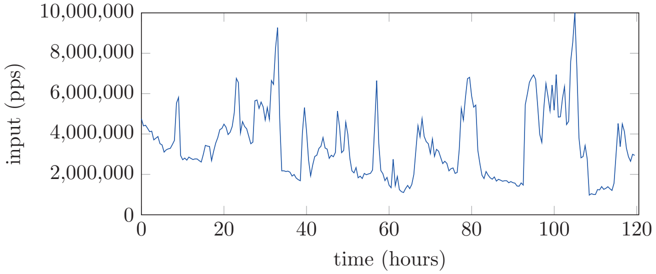


Figure 3.7 Traffic data used for the evaluation in this chapter, gathered from the Swedish University Network (SUNET).

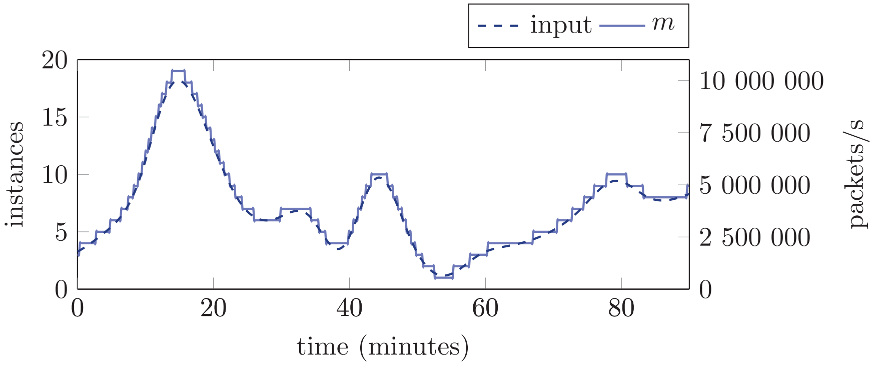


Figure 3.8 Simulation data showing how the node scales the number of virtual machine instances over time.

Example node

For this example a single node was simulated, where the parameters were randomly generated. The deadline $\mathscr{D}$ was chosen, randomly, from the interval $[0.05,\ 0.10]$ seconds. The nominal service-rate $\bar{s}$ was chosen uniformly at random from the interval $[300000,\ 600000]$ packets per second, and the time-overhead Δ was drawn uniformly at random from the interval $[15,\ 60]$ seconds. The incoming traffic $r(t)$ was used for this simulation was chosen from a randomly selected 90min-interval of the SUNET-data shown in Figure 3.7. This was then scaled in order to have a peak of 10000000 packets per second.

In Figure 3.8 one can see how the arrival rate $r(t)$ varies over time, and how the node adjusts the number of active virtual machines in order to compensate for this. This can also be seen in Figure 3.9 where the maximum processing capacity $s^{\text{cap}}(t)$ of the nodes closely follows the arrival rate. There, one can also see how the admission rate is adjusted. It is also interesting to note how the utility $u(t)$ changes over time during the simulation. Although it remains high during most of the simulation, it does dip a bit around the 50 minute mark. One reason for this is that the arrival rate is quite low relative to the nominal service rate $\bar{s}$. This leads to only a few virtual machine being needed, and thus a relatively larger difference between $r(t)$ and $s^{\text{cap}}(t)$.

Comparing AutoSAC with state-of-the-art

In this section compare AutoSAC with two “industry”-methods through a Monte Carlo simulation where each method is simulated a 1000 times. The two industry-methods are; *dynamic auto-scaling* (DAS) and *dynamic over-allocation* (DOA). Since neither of these two industry methods use any admission control they are also augmented with the admission controller presented in Section 3.2. The two augmented methods are denoted by “DAS AC” and “DOA AC”.

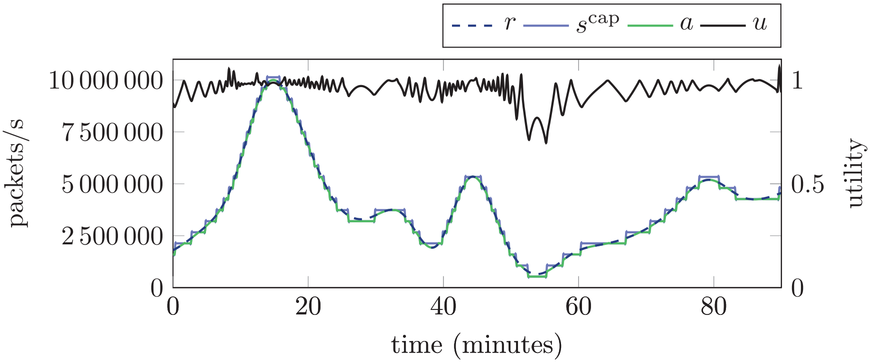


Figure 3.9 Simulation data showing how the arrival rate $r(t)$, maximum processing capacity $s^{\text{cap}}(t)$, admission rate $a(t)$ and the utility $u(t)$ varies over time.

Dynamic auto-scaling (DAS) This method is currently being offered to customers using Amazon Web Services. It allows the user to monitor different metrics (e.g., CPU utilization) of their VMs using CloudWatch. One can then use it together with their auto-scaling solution to achieve *dynamic auto-scaling*. This allows the user to scale the number of VMs as a function of these metrics. One should note that the CPU utilization can be considered the same as the efficiency metric $u^{\mathrm{e}}(t)$ defined in (3.7). For the Monte Carlo simulation the following rules were used:

$$\begin{cases} \text{add a virtual machine} & \text{if } u^{\mathrm{e}}(t) \geq 0.9, \\ \text{remove a virtual machine} & \text{if } u^{\mathrm{e}}(t) \leq 0.8. \end{cases}$$

Dynamic over-allocation (DOA) A downside with DAS is that it reacts slowly to sudden changes in the input. A natural alternative would therefore be to instead do *dynamic over-allocation*, where one measures the input to each node and allocates virtual machines such that there is an expected over-provisioning of the processing capacity of 10%.

Monte Carlo Simulation The five methods are compared using a Monte Carlo simulation with 1000 runs for each method. For every run, 2 hours of the input data was randomly selected from the total of 120 hours shown in Figure 3.7. Furthermore, in every run a new node was generated using the method described earlier.

Results The mean of the average utility over all the simulation runs is presented in Figure 3.10 for each of the five methods. One can see that AutoSAC achieves a utility that is 25–30% better than that of DAS and DOA. The main reason for this is that they are lacking admission control leading to packets missing their deadlines, which eventually results in a low utility.

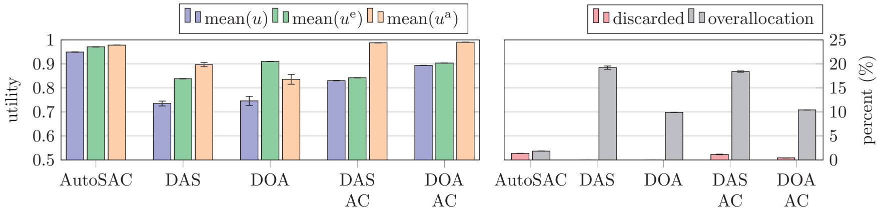


Figure 3.10 Results from the Monte Carlo simulation. AutoSAC was compared against dynamic auto-scaling (DAS) and dynamic overallocation (DOA). Neither DAS nor DOA has admission control so they were augmented with the one developed in this chapter. In the left figure one can see the result of the average utility, efficiency, and availability for each of the five methods, and in the right figure the fraction of the incoming packets that are discarded and the average amount of overallocation of the processing capacity.

When augmenting DAS and DOA with the admission controller presented in Section 3.2 their performance is increased by about 20%, purely as a result of not having these sudden drops in performance. However, AutoSAC still performs better, due to a higher efficiency, the main reason for this is likely because of better estimation of how many virtual machine are needed to processing the incoming traffic.

3.4 Summary

In this chapter we have developed a mathematical model for a virtual network function, or a cloud service. The model captures, among other things, the time needed to start/stop virtual resources (e.g., virtual machines or containers). The main idea behind the suggested controllers is to create an abstraction between: i) predicting future arrival rates, ii) controlling how many virtual machines will be needed, and iii) computing how much traffic should be admitted into the node.

To evaluate the performance of the suggested method, AutoSAC was compared against two standard-methods in industry for controlling cloud resources. The evaluation highlights the importance of good admission control if one wish to have both a high efficiency of the utilized resources yet still guarantee that a deadline for the arriving traffic is met.

The method presented in this chapter will be generalized in Chapter 6 in order to control a chain of connected nodes.

4

HoloScale – combining horizontal and vertical scaling

In this chapter we go on a tangent compared to the previous chapter. We still consider a single cloud node, but instead of combining admission control and horizontal scaling to control the response-time of the node, we treat a different problem. We aim at combining horizontal and vertical scaling, but no admission control, in order to only control the available processing capacity of a node. The proposed method is called HoloScale, and we will illustrate how it could be combined with a response-time controller in order to ensure that the response-time of a node remains below a given deadline. In Figure 4.1 we illustrate the idea of combining horizontal and vertical scaling.

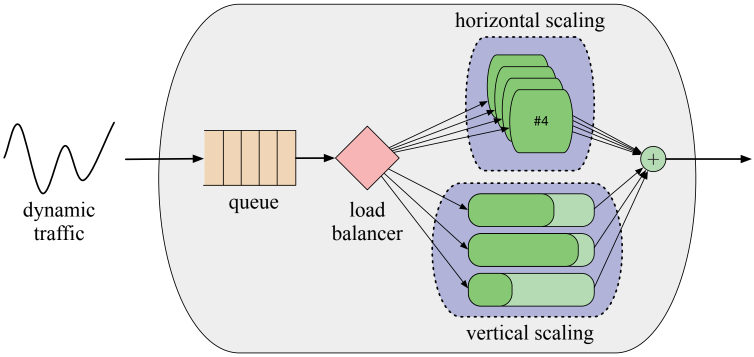


Figure 4.1 Illustration of the how horizontal and vertical scaling are combined in HoloScale. The goal is to ensure that the available processing capacity of the node matches the desired one.

4.1 Introduction

Holo originates from the Greek work $o\lambda o\varsigma$ meaning “whole, entire, complete” as well as “safe and sound”. As we saw in the previous chapter, this is typically not the case when using solely horizontal scaling for controlling the processing capacity of a cloud node. When scaling cloud resources, one typically has to choose between one of two options: a) *horizontal scaling*, or b) *vertical scaling*. Horizontal scaling, is where one adds processing capacity by adding more instances which are identical. The benefit of which is that one can scale the processing capacity of a node over a very large range. The downside, however, is that it is usually very slow and with a very coarse control. The other option, vertical scaling allows the user to quickly and continuously scale the existing processing capacity of a node by adding more resources to an existing, and already running, machine. The downside, however, is that the operating range is limited.

With HoloScale we use basic principles from control-theory and present a way to combine both horizontal and vertical scaling in a provably safe way. This allows us to do both rapid continuous scaling (by leveraging vertical scaling) over a large spectrum (by leveraging horizontal scaling). We present a model for this system and solve the control problem by borrowing ideas and concepts from classical mid-range control. Finally, we evaluate our proposed solution through simulations where the set-point for the processing capacity is gathered from the SUNET-data presented in Chapter 3. We show that the system is capable of quickly scaling the computation resources as well as to react to sudden disturbances, such as virtual machines going down.

A cascaded control With the advent of cloud computing it has become possible to autonomously scale the resources allocated to various cloud services. The goal is to ensure some quality of service (QoS) for the service. For instance, this could for instance be to ensure a specific response-time when using the service, or that there is a sufficiently high throughput. Naturally, there is no engineering challenge in “just” ensuring that this QoS is met, since one option is to always over-dimension the amount of resources allocated to the service. The challenge therefore, lies in providing “lagom” (a popular Swedish for “just enough”) amount of resources to allocate such that the QoS is met, but with the lowest possible cost.

When controlling such a cloud computing system, there is typically a need to use a feedback-loop, as illustrated in Figure 4.2, (and as we saw in Chapter 3) since the true computing capacity might be subject to disturbances. Such disturbances could be virtual machines or containers crashing, or bugs in the software leading to a lack of performance. This could lead to the fact that the actual processing capacity of the system differs from the desired one, thereby making it difficult for the QoS-controller to ensure that the QoS-goals are met.

We believe that it would be simpler to design these QoS-controllers if the design-engineer could assume that the actual processing capacity of the system

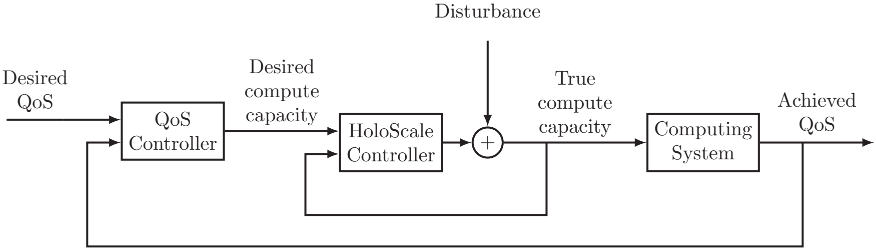


Figure 4.2 Illustration of the cascaded feedback-loops allowing a separation of concerns. The HoloScale controller ensures that the true processing capacity matches the desired processing capacity, thereby allowing the QoS-controller to focus controlling the desired processing capacity such that the QoS-goals are met.

matches the one which is desired from the QoS-controller. If this would be possible, the engineers would be allowed to solely focus on the dynamics and mechanisms that affect the QoS, rather than to also handle the situation when there are large changes of the desired compute capacity, or when machines crash.

One way to address this is to use a cascade-controller, as illustrated in Figure 4.2. One part of the controller has the focus on ensuring that the system has the desired amount of processing capacity (the HoloScale-controller), while the other part (the QoS-controller) has the focus on adjusting the amount of processing capacity such that the QoS-goal is met.

4.2 System model

In this section we describe some of the underlying assumptions as well as the system model we used to derive the control structure.

The goal of the system is to ensure that the *true processing capacity* $s^{\mathsf{cap}}(t) \in \mathbb{R}^+$ matches the *desired processing capacity* $s^{\mathsf{ref}}(t)$. We assume that we have one set of machines capable of vertical scaling as well as one set of machines which we can scale horizontally. To distinguish between the two we will call the capacity provided by the vertically scalable machines as the *vertical capacity* $s_v(t) \in [s_v^{\mathsf{lb}}, s_v^{\mathsf{ub}}]$, where $s_v^{\mathsf{lb}} \in \mathbb{R}^+$ and $s_v^{\mathsf{lb}} \leq s_v^{\mathsf{ub}} \in \mathbb{R}^+$ are the lower and upper limit of the vertical scaling capabilities. Similarly, the capacity provided by the horizontally scalable system is denoted as the *horizontal capacity* $s_h(t) = \bar{s} \cdot m(t)$, where $m(t) \in \mathbb{N}$ is the number of instances that are currently running, and $\bar{s} \in \mathbb{R}^+$ is the processing capacity for one instance. Together, the horizontal and the vertical capacity form the *total capacity* for the system, $s^{\mathsf{cap}}(t) = s_v(t) + s_h(t)$.

It is possible to change the vertical capacity through a control signal $s_v^{\mathsf{ref}}(t)$, but doing so takes some time Δ_v, giving us the relationship $s_v(t) = s_v^{\mathsf{ref}}(t - \Delta_v)$. Similarly, it is possible to control the horizontal capacity by controlling the number of instances we wish to have running through $m^{\text{ref}}(t)$. Performing this action also takes some time Δ_h, leading to the horizontal capacity at time t being given by $s_h(t) = \bar{s} \cdot m(t) = \bar{s} \cdot m^{\text{ref}}(t - \Delta_h)$. We assume that changing the horizontal capacity is much slower than changing the vertical capacity, i.e., $\Delta_v \ll \Delta_h$.

A chaos monkey In real life the number of instances you have running does not always match the number of instances you thought you had running. The reason is that sometimes, machines crash or does not boot up correctly. In order to simulate this in our set-up we used inspiration from the field of chaos engineering [Basiri et al., 2016; Chang et al., 2015; Beyer et al., 2016], and added a *Chaos Monkey* to our system. The job of it is to sometimes kill some of the instances running, thereby reducing the horizontal capacity. The number of *killed* instances is given by $\varepsilon(t)$, leading to the following horizontal capacity: $s_h(t) = \bar{s} \cdot \big(m^{\text{ref}}(t - \Delta_h) + \varepsilon(t)\big)$.

4.3 HoloScale controller

The main idea behind the controller structure comes from the classic mid-range problem in control theory [Åström and Murray, 2010]. The basic idea is to separate the concerns for the horizontal and the vertical controller. In doing so we can allow the vertical controller to focus on ensuring that the current processing capacity of the system matches the desired processing capacity. However, since the range-of-operation for this controller is very limited, we must ensure that it remains within its operating range. This is the purpose of the horizontal controller: to match the desired processing capacity in a coarse manner, so that the vertical controller can be able to fill in the void.

The basic controller-structure can be seen in Figure 4.3, which illustrated that the horizontal controller is a feedforward controller from the desired processing capacity to the number of virtual machines which should be running. The vertical controller is a feedback-loop aiming to ensure that the true processing capacity matches the desired processing capacity. They will both be described in more detail later on.

As mentioned earlier, our approach is similar to that of mid-range control problems, but it does differ slightly. The main difference is that we do not use feedback from the vertical controller back to the horizontal controller. The reason for this is that the horizontal controller operates in a discrete space (choosing the number of instances that should be running), while the vertical controller operates in a continuous space. Thus, if having feedback from the vertical controller to the horizontal controller the system will end-up in a state where the horizontal controller tries to drive its processing capacity so that it matches a continuous signal. This will be achieved by scaling up/down the number of running instances very rapidly, much like with pulse-width modulation. Since we assume that there is a cost associated with starting a virtual machine this behavior is not desirable, hence we skip this

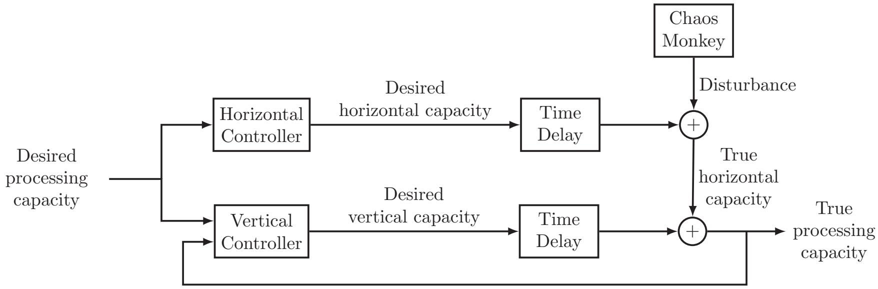


Figure 4.3 Overview of the controller structure for combining a horizontal and vertical scaler. As one can see, there is an open-loop structure from the reference signal, through the horizontal controller and a feedback loop from the total processing capacity back to the vertical autoscaler.

feedback. Another reason for omitting the feedback is because the time-delay for adding/removing a virtual machine is orders of magnitude slower than the vertical controller. Hence, from the point-of-view of the vertical controller, the control action from the horizontal controller will look a lot like an open-loop feedforward controller.

The horizontal controller

As mentioned earlier, the job of the horizontal controller, illustrated in Figure 4.4, is to ensure that the vertical controller is within its working range. To achieve this, it takes as input the desired maximum processing capacity $r(t)$ and computes how many instances it needs to have running in order to match the floor of this:

$$s_h^{\mathsf{ref}}(t) = m^{\mathsf{ref}}(t) \cdot \bar{s} = \left\lfloor \frac{r(t)}{\bar{s}} \right\rfloor \cdot \bar{s}. \tag{4.1}$$

As mentioned earlier, some machines might crash during run-time, so it is important to still be able to ensure that the number of machines that are up-and running matches the desired number. To achieve this, the horizontal controller has an additional feedforward-loop, from the disturbance signal $\varepsilon(t)$ to the desired horizontal capacity $s_h^{\mathsf{ref}}(t)$. This is in order to compensate for any crashed machines. With $\varepsilon(t)$ machines being down at time t it has to compensate for a total of $\bar{s} \cdot \varepsilon(t)$. The complete controller structure is illustrated in Figure 4.4.

The vertical controller

The vertical controller, shown in Figure 4.5, is a simple proportional and integral controller, or PI-controller, which takes as input the error, i.e., the difference between the desired processing capacity and the true processing capacity of the sys-

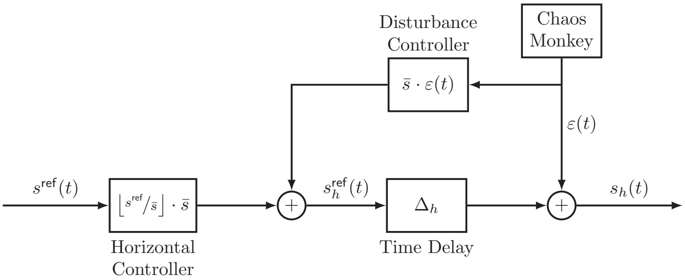


Figure 4.4 Block diagram for the horizontal controller. It consists of two feedforward controllers, from the desired processing capacity as well as from the measured disturbance, i.e., the number of crashed machines.

tem, and computes the necessary vertical processing capacity:

$$s_v^{\text{ref}}(t) = k_p \cdot e(t) + k_i \int_0^t e(x)\mathrm{d}x, \tag{4.2}$$

where $e(t) = s^{\text{ref}}(t) - s^{\text{cap}}(t)$ is the control-error, or the difference between desired and true processing capacity, and where k_p and k_i are the control-parameters deciding the "weights" of the proportional and integral part of the controller. The vertical processing capacity, is then a time-delayed version of this:

$$s_v(t) = s_v^{\text{ref}}(t - \Delta_v).$$

The question of how to choose these control-parameters k_p and k_i will depend on the how the user wishes the final system to behave. For instance, if the user wishes the system to react quickly to changes in desired processing capacity, or if the user wishes the system to not have any *overshoot*, where overshoot means that the system for a brief moment will add slightly too much processing capacity only to remove shortly after.

We will discuss this in the next section, where we will also derive necessary conditions on what k_p and k_i can be chosen and still guarantee that the system will remain stable.

4.4 Design principles and stability analysis

Whenever designing a controller it is very important to be sure that the controller is capable of stabilizing the intended system, even in the presence of disturbances and modeling errors. In this section we will derive conditions for how to choose the control-parameters k_p and k_i so that the system will remain stable.

One of the benefits when combining the horizontal feedforward controller with the vertical feedback controller is that a feedforward-loop can never destabilize a

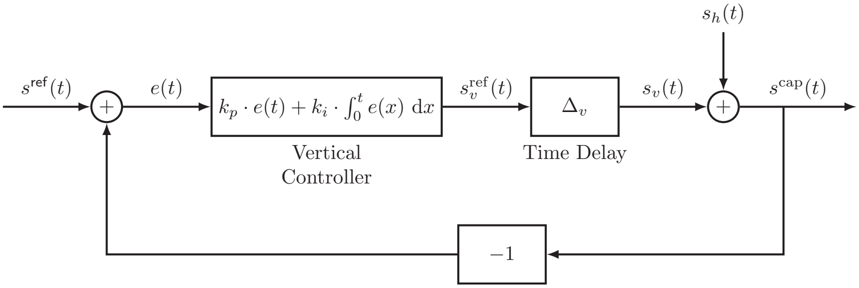


Figure 4.5 Block diagram for the vertical controller. It uses feedback information from the total processing capacity in order to compute how much "extra" is needed from the vertical controller.

stable control-loop [Åström and Murray, 2010]. For our purposes, this means that we only have to consider the vertical controller when deriving the parameters for which it is stable. From the perspective of the vertical controller, the processing capacity provided by the horizontal controller can be viewed as an external disturbance.

There are many ways to analyze a control-system and derive conditions on its stability. In this section we will use the Nyquist Criterion [Åström and Murray, 2010; Nyquist, 1932]. Briefly described, it can determine the stability of a feedback system by analyzing the loop transfer function $G(s)$, where $s \in \mathbb{C}$ is a complex number and not be confused with the processing capacity (whenever we refer to the processing capacity, it will always be written as a function of time: $s(t)$). Simplified, the condition states that as long as there are no poles in the closed right half-plane, the closed-loop system is stable if and only if the closed contour given by $\Omega = \{G(i\omega) : -\infty < \omega < \infty\} \subset \mathbb{C}$ has no net encirclements of the critical point $s = -1$.

In our case, the Nyquist criterion is fulfilled by ensuring that for the frequency ω_c giving us $|G(i\omega_c)| = 1$, it hold that $\arg\big(G(i\omega_c)\big) > -\pi$, and that for the frequency ω_0 giving us $\arg\big(G(i\omega_0)\big) = -\pi$, it hold that $|G\big(i\omega_0\big) < 1$. The loop-transfer function in our case is given by the Laplace transform of the vertical controller combined with the time-delay. That is, the Laplace transform of $g(t) = s_v(t) = s_v^{\text{ref}}(t - \Delta_v)$:

$$G(s) = \mathscr{L}\{g(t)\} = \frac{sk_p + k_i}{s} e^{-s\Delta_v}. \tag{4.3}$$

We begin our analysis by splitting $G(s)$ in two parts:

$$G(s) = G_0(s) \cdot e^{-s\Delta_v}, \quad G_0(s) = \frac{sk_p + k_i}{s}$$

We can then find the first condition on $k_p \geq 0$ and $k_i \geq 0$ by finding for which values of them it holds that $|G(i\omega_0)| < 1$. This leads to the following condition:

$$|G(i\omega_0)| = |G_0(i\omega_0)| = \frac{|i\omega_0 k_p + k_i|}{|i\omega_0|} < 1.$$

From the above expression we can find the maximum value of k_p by setting $k_i = 0$:

$$k_i = 0 \quad \Rightarrow \quad 0 \leq k_p < 1. \tag{4.4}$$

The next step is then to check for which values of k_i and $k_p < 1$ it hold that $\arg\big(G(i\omega_c)\big) > -\pi$. We therefore begin by locating the cross-over frequency ω_c:

$$\omega_c \in \mathbb{R}^+ \ : \ |G_0(\omega_c) \cdot e^{-\omega_c \Delta_v}| = 1,$$

and then ensure that

$$\arg(G_0(\omega_c) \cdot e^{-\omega_c \Delta_v}) > -\pi.$$

The cross-over frequency ω_c can be located from:

$$|G_0(\omega_c) \cdot e^{\omega_c \Delta_v}| = |G_0(\omega_c)| = \left| \frac{i\omega_c k_p + k_i}{i\omega_c} \right| = \frac{\sqrt{\omega_c^2 k_p^2 + k_i^2}}{\omega_c} = 1,$$

giving us the following condition:

$$\sqrt{\omega_c^2 k_p^2 + k_i^2} = \omega_c \quad \Rightarrow \quad \omega_c = \frac{k_i}{\sqrt{1 - k_p^2}}. \tag{4.5}$$

For this frequency, the argument $\arg\big(G(i\omega_c)\big)$ is given by

$$\begin{aligned} \arg\big(G_0(\omega_c) \cdot e^{-\omega_c \Delta_v}\big) &= \arg\big(G_0(\omega_c)\big) - \omega_c \Delta_v \\ &= \arg\left(\frac{i\omega_c k_p + k_i}{i\omega_c}\right) - \omega_c \Delta_v \\ &= \arg\Big(i\omega_c k_p + k_i\Big) - \frac{\pi}{2} - \omega_c \Delta_v \\ &= \arctan\left(\frac{\omega_c k_p}{k_i}\right) - \frac{\pi}{2} - \omega_c \Delta_v. \end{aligned}$$

This gives us the following condition:

$$\arctan\left(\frac{\omega_c k_p}{k_i}\right) - \frac{\pi}{2} - \omega_c \Delta_v > -\pi. \tag{4.6}$$

Finally, by inserting (4.5) into (4.6) we arrive at

$$\begin{aligned} &\arctan\Big(\tfrac{k_p}{\sqrt{1-k_p^2}}\Big) - \tfrac{\pi}{2} - \tfrac{k_i}{\sqrt{1-k_p^2}} \Delta_v > -\pi \\ \Rightarrow \quad &k_i < \tfrac{\sqrt{1-k_p^2}}{\Delta_v} \cdot \Big(\tfrac{\pi}{2} + \arctan\big(\tfrac{k_p}{\sqrt{1-k_p^2}}\big)\Big). \end{aligned}$$

The conditions on the control parameters k_p and k_i can then be summarized by:

$$\begin{cases} k_p \in [0,\, 1), \\ k_i < \frac{\sqrt{1-k_p^2}}{\Delta_v} \cdot \left(\frac{\pi}{2} + \arctan\left(\frac{k_p}{\sqrt{1-k_p^2}} \right) \right). \end{cases} \tag{4.7}$$

Choosing control parameters With the stability conditions for k_p and k_i given in Equation (4.7) it becomes possible to compute a region of stable control parameters. This means that any choice of k_p and k_i from within this region will lead to a stable controller. Naturally, the size of this region depends on the time-delay Δ_v before the control action of the vertical controller is realized. Since $0 \leq k_p < 1$ it is possible to fix a value of $k_p \in [0,\, 1)$ and then use Equation (4.7) to compute for which $0 \leq k_i$ the system will remain stable. One can then continue and repeat this while sweeping over $k_p \in [0,\, 1)$. In Figure 4.6 we have done this and plotted the stable regions of k_p and k_i for two different time-delays: $\Delta_v = 0.10$ and $\Delta_v = 0.20$.

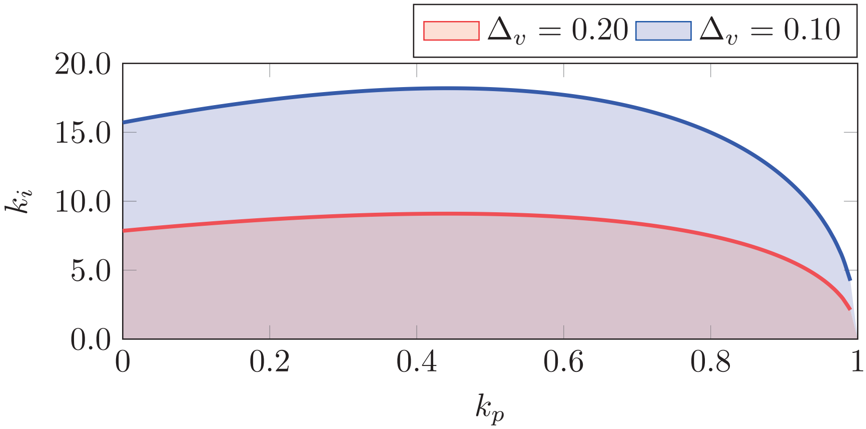


Figure 4.6 Regions for stable choices of k_p and k_i given different time-delays for the vertical controller, i.e., Δ_v. The red region represent stable values for $\Delta_v = 0.2$ s, and the blue region for $\Delta_v = 0.10$ s.

Anti-windup for the vertical controller Since the vertical controller has a limited range, and uses an integrator, it is important to ensure that it does not become subject to *integral wind-up*. This would for instance occur whenever the vertical controller tries to achieve a capacity higher than its upper bound: $s_v^{\text{ref}}(t) > s_v^{\text{ub}}$. To remedy this problem, one can add an anti-windup solution to the controller. One way to do this is to feed back the difference between the control signal fed into the system, $s_v^{\text{ref}}(t)$ and the saturated signal s_v^{ub}, as shown in [Åström and Murray, 2010].

Prediction for the horizontal controller Since the horizontal controller is slow to react to changes of the desired capacity it might be advantageous to use *prediction*, just as we did in Chapter 3. This means that it tries to predict what the computation requirement will be in Δ_h time-units into the future with a simple linear extrapolation, such that

$$\hat{s}^{\mathsf{ref}}(t) = s^{\mathsf{ref}}(t) + \Delta_h \cdot \frac{\partial \tilde{s}^{\mathsf{ref}}(t)}{\partial t}, \tag{4.8}$$

where $\tilde{s}^{\mathsf{ref}}(t)$ is a low-pass filtered version of $s^{\mathsf{ref}}(t)$.

4.5 Evaluation

To evaluate the performance of the HoloScale-controller, we performed many simulations, again using the SUNET-data presented in Chapter 3 as a reference for the desired processing capacity of the node. HoloScale was compared with using just a horizontal controller. For the purely horizontal controller the number of virtual machine instances used was chosen such that $m^{\mathsf{ref}}(t) = \lfloor s^{\mathsf{ref}}(t)/\bar{s} \rceil$. The performance of the two methods was evaluated using the *average normalized integrated absolute error* (avg IAE) as performance metric:

$$\text{avg IAE}(t) = \frac{1}{t} \int_{0}^{t} \frac{|s^{\mathsf{ref}}(x) - s^{\mathsf{cap}}(x)|}{s^{\mathsf{ref}}(x)} \, \mathrm{d}x. \tag{4.9}$$

Example simulation For every simulation the time-delay for horizontal scaling was chosen to be $\Delta_h = 1$ s and the time-delay for vertical scaling was chosen to be $\Delta_v = 0.1$ s. The peak traffic intensity was again 10 000 000 packets per second. The nominal service capacity for a horizontal virtual machine was randomly and uniformly chosen from the interval $\bar{s} \in [833,\ 125]$ in order to ensure that the maximum number of horizontal instances needed would be less than $8 - 12$. The capacity for the virtually scalable machine was chosen to be $s_v^{\mathsf{ub}} = 2.5 \cdot \bar{s}$. Finally, the control-parameters for the HoloScale-controller was chosen to be $k_p = 0.25$ and $k_i = 7.5$.

In Figure 4.7 we illustrate one of the simulations of where we compare HoloScale and the horizontal controller. One can see that the Chaos Monkey killed a horizontal virtual machine at times $t = \{20, 40, 60, 80\}$. The difference between the two figures however, is that in HoloScale, the vertical controller immediately compensated for this. When only having horizontal control, it took $\Delta_h = 1$ s before this shortage of processing capacity compensated for.

In Figure 4.8 we show the horizontal capacity $s_h(t)$ and the vertical capacity $s_v(t)$ for the HoloScale controller during the interval $t = [0, 50]$ in order to illustrate

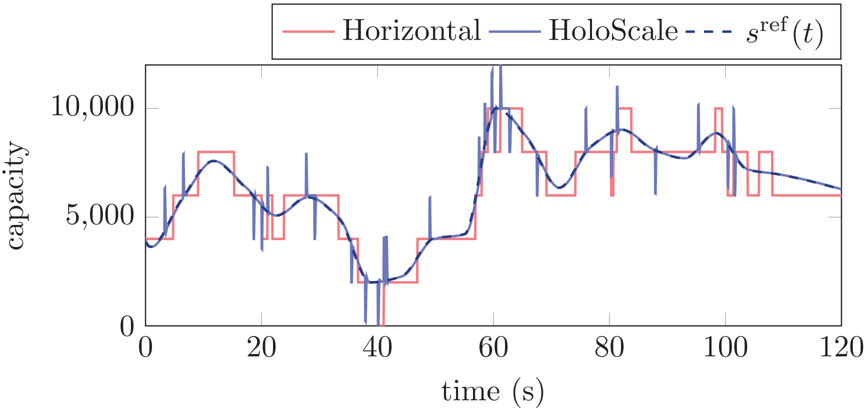


Figure 4.7 Total processing capacity when combining horizontal and vertical scaling as proposed in the HoloScale controller.

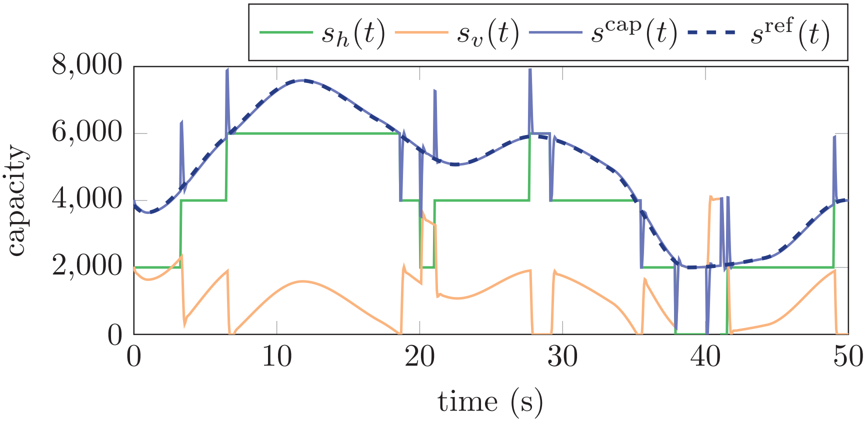


Figure 4.8 Horizontal and vertical processing capacity in the HoloScale controller during the simulation. Note that the Chaos Monkey killed a virtual machine image at times $t = \{20, 40\}$, and that the vertical controller quickly compensated for this.

the interaction between the two a bit more clearly. There, one can see that the vertical controller quickly compensates for any lack of processing capacity provided by the horizontal controller.

Monte Carlo simulation To compare the HoloScale against just horizontal scaling, we performed a small Monte Carlo simulation where each method was simulated 100 times each. For each simulation the set-point for the desired processing capacity was chosen to follow the SUNET-data, just as illustrated in Figure 4.7. We then evaluated the mean avg IAE for both controllers. The result is shown in Figure 4.9, which illustrates how mean(avg IAE) changes over time. One can especially see the effects of the Chaos Monkey killing a virtual machine instance at times $t = \{20, 40, 60, 80\}$, and that HoloScale was able to compensate for this disturbance quicker. This in-turn lead to the avg IAE of HoloScale being only 25% of the value obtained when one is only using a horizontal controller.

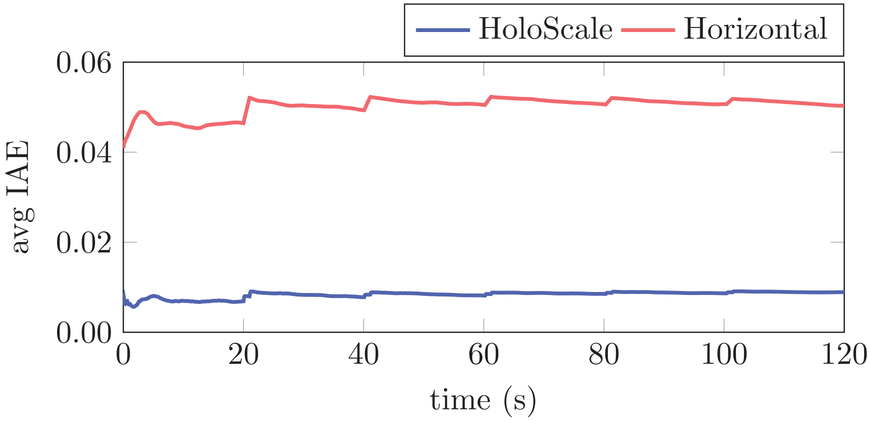


Figure 4.9 Plot showing the results of the Monte Carlo simulation where we compare HoloScale against a pure horizontal controller. The y-axis shows the average integrated absolute error over all the simulations.

4.6 Summary

In this chapter we have presented an alternative method to control the processing capacity of a node. The idea is that if one can combine both horizontal scaling and vertical scaling one can end up with a smoother control of the available processing capacity. This makes it easier to then control the computation-system in order to reach a desired quality-of-service of the system.

To combine these two scaling mechanism we have borrowed ideas from the classical mid-ranging control problem, and showed how one can guarantee that the proposed controller is stable. Finally we evaluated the performance of the proposed HoloScale through a number of simulations where it was compared with a purely horizontally scalable system. It should be noted that this chapter is more of a theoretical comment, since it is currently quite difficult to scale down resources using vertical scaling, due to difficulty of reducing memory during run-time and the need to adjust applications to automatically scale with more/less resources.

The proposed method in this chapter might allow one to abandon the admission controller in AutoSAC, and instead use a cascaded QoS-controller, where the outer loop controls the response-time, or the latency, of the node and the inner loop ensures that the processing capacity matches the desired one and is controlled by the HoloScale controller. This would indeed be an interesting direction for future work.

5

A naive approach for controlling a chain of nodes

In this chapter, we are back on the path towards controlling a network of nodes in order to ensure that the traffic moving through it will meet their end-to-end deadlines. The focus in this chapter is to study how to ensure that traffic flowing through a chain of nodes can be guaranteed to meet their end-to-end deadline. To achieve this every node in the chain will be able to horizontally scale the number of active virtual machines, or containers, in order to dynamically adjust their processing capacity, as illustrated in Figure 5.1. As a first encounter to this problem, will keep the incoming traffic rate of the chain constant, so as to simplify the problem. Later, in Chapter 6 we will extend this to also allow for dynamic traffic-rates.

The key contributions of this chapter is that with a constant traffic intensity, we can set up an optimization problem and derive a schedule for turning the virtual machines on/off. At the end of the chapter, we will show how adding a feedback-law will allow the nodes to also handle stochastic variations to either their available processing capacity, or to the incoming traffic intensities.

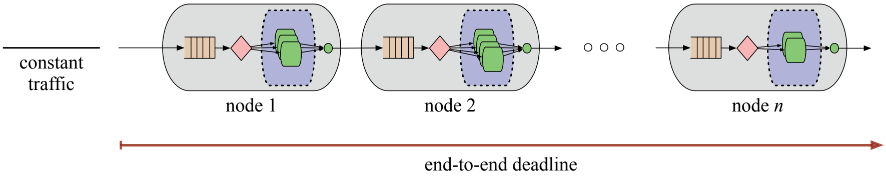


Figure 5.1 Illustration of the chain of cloud services (or virtual network functions) considered in this chapter. The traffic flowing through the n nodes must be processed by all of the nodes within a specific end-to-end deadline $\mathscr{D}$. Note that the nodes considered in this chapter only rely on horizontal scaling, and no vertical scaling nor the use of any admission control.

5.1 Modeling a chain of nodes

We model the chain of nodes as the set $\mathscr{V} \in \mathbb{N}^+$, with a total of $n = |\mathscr{V}|$ nodes, which are concatenated as is illustrated in Figure 5.1. This means that the output of node 1 is the input of node 2, and so on. In order to keep track of the different nodes in the chain we will use the sub-index i to specify the properties of that particular node. This means that the model presented in Chapter 3 is simply extended, such that for the i-th node $r_i(t)$ is the traffic entering it, $m_i(t)$ is the number of virtual machines, and $s_i(t)$ is the service rate and the rate by which traffic is leaving the node. This also means that for nodes $i = 2, \dots, n$ it follows that $r_i(t) = s_{i-1}(t)$, implying that we do not consider any communication latency between the nodes, nor any packet loss during the communication between the nodes. It should be noted that just as the case with processing time in Chapter 3, a communication delay between nodes could be considered by properly adjusting the end-to-end deadline $\mathscr{D}$. For instance, if the total communication delay between all the nodes in the chain adds up to c, then the adjusted end-to-end deadline could be given by $\tilde{\mathscr{D}} = \mathscr{D} - c$.

Different nodes in the chain are assumed to provide different processing to the traffic passing through them. For instance, one node might be a firewall, another one might be a deep packet inspector, a control application or a machine learning inference application. Therefore it is assumed that the *nominal processing rate* of a virtual machine instance might be different depending on which node it belongs to. Therefore, we will also use a subscript index to capture this leading to $\bar{s}_i$ being the nominal processing rate for a virtual machine instance belonging to node i. This implies that the maximum processing capacity of node i at time t is given by $s_i^{\text{cap}}(t) = m_i(t) \cdot \bar{s}_i(t)$, (here it should be noted again that we do not assume any machine uncertainty as presented in Chapter 3).

As presented in Chapter 3 it is useful to define the cumulative traffic flowing through a node, and for the i-th node, this will be defined as:

$$S_i(t) = \int_0^t s_i(x)\,\mathrm{d}x, \qquad R_i(t) = \int_0^t r_i(x)\,\mathrm{d}x. \tag{5.1}$$

Moreover, since we do not have any admission control in this chapter, we can model the amount of work (or traffic) in the queue of the i-th node as:

$$q_i(t) = R_i(t) - S_i(t). \tag{5.2}$$

Response time and latency. In order to properly define the end-to-end response time or latency experienced by traffic flowing through the chain of nodes, we begin by defining the latency of the i-th node as $L_i(t) \in \mathbb{R}^+$:

$$L_i(t) = \inf\{\tau \geq 0 : R_i(t-\tau) \leq S_i(t)\}. \tag{5.3}$$

It is also possible to define the end-to-end latency in a similar way, but by instead comparing $S_n(t)$ with $R_1(t-\tau)$, since there is assumed to be no packet loss between the nodes in the chain and no communication delay. Therefore, the latency experienced by a packet passing through all the nodes from 1 thru i can be given by:

$$\mathscr{L}_i(t) = \inf\{\tau \geq 0 : R_1(t-\tau) \leq S_i(t)\}. \tag{5.4}$$

Finally, the end-to-end latency for passing through the entire chain is given by $\mathscr{L}(t) = \mathscr{L}_n(t)$.

Cost model

To be able to provide guarantees about the behavior of the service chain, it is necessary to make *hard reservations* of the resources needed by each node in the chain. This means that when a certain resource is reserved, it is guaranteed to be available for utilization. Reserving this resource results in a cost, and due to the hard reservation, the cost does not dependent on the actual utilization, but only on the resource reserved.

Compute cost. The *computation cost* per time-unit per machine is denoted j_i^{c}, and can be seen as the cost for the CPU-cycles needed by one machine instance in the i-th node. Naturally, this means that we assume that all the machine instances with a single node i are identical in cost. This cost will also charged during the time-overhead Δ_i. Without being too conservative, this time-overhead can be assumed to occur only when a machine is started. The *average computing cost* per time-unit for the i-th node is therefore given by

$$J_i^{\mathsf{c}}(m_i) = \lim_{t\to\infty} \frac{j_i^{\mathsf{c}}}{t} \int_0^t m_i(s) + \Delta_i \cdot \left(\partial^- m_i(s)\right)_+ \mathrm{d}s \tag{5.5}$$

where $(x)_+ = \max(x, 0)$, and $\partial^- m_i(t)$ is the left-limit of $m_i(t)$:

$$\partial^- m_i(t) = \lim_{x\to t^-} \frac{m_i(t) - m_i(x)}{t-x},$$

that is, a sequence of Dirac's deltas at all points where the number of machines changes. This means that the value of the left-limit of $m_i(t)$ is only adding to the computation-cost whenever it is positive, i.e. when a machine is switched on.

Buffer cost. The *buffer cost* per time-unit per space for a request is denoted j_i^{q}. This can be seen as the cost that comes from the fact that physical storage needs to be reserved such that a queue can be hosted on it. Normally this would correspond to the RAM of the network-card. Each node is assumed to have a fixed *maximum buffer-capacity* $q_i^{\max} \in \mathbb{R}^+$, representing the largest number of requests that can be stored in the i-th node. Reserving the capacity of $q_i^{\max}$ would thus result in a cost per time-unit of

$$J_i^{\mathsf{q}}(q_i^{\max}) = j_i^{\mathsf{q}} \cdot q_i^{\max}. \tag{5.6}$$

Problem formulation

The aim in this chapter is to control the number $m_i(t)$ of virtual machine instances running in the i-th node, such that the average cost is minimized. However, possible solutions are constrained by the fact that the end-to-end latency $\mathscr{L}(t)$ can never exceed the end-to-end deadline $\mathscr{D}$. Moreover, the maximum queue sizes q_i^{max} cannot be exceeded either. Together, this can be posed as the following problem:

$$\begin{aligned} &\text{minimize} && J = \sum_{i\in\mathscr{V}} J_i^{\mathsf{c}}\big(m_i\big) + J_i^{\mathsf{q}}(q_i^{\mathsf{max}}) && \\ &\text{subject to} && \mathscr{L}(t) \leq \mathscr{D} && \forall t \geq 0 \\ & && q_i(t) \leq q_i^{\mathsf{max}}, && \forall t \geq 0, \quad i \in \mathscr{V} \end{aligned} \tag{5.7}$$

with J_i^{c} and J_i^{q} as in (5.5) and (5.6), respectively. In this chapter the optimization problem (5.7) will be solved for a service-chain fed with a constant incoming rate $r(t) = r$. Later on a feedback-law will be derived allowing the system to remain stable under a stochastic input.

A valid lower bound J^{lb} to the cost achieved by any feasible solution of (5.7) is found by assuming that all nodes are capable of providing exactly a service rate r equal to the input rate. This is possible by running a fractional number of machines $r/\bar{s}_i$ at the i-th node. In such an ideal case, buffers can be of zero size ($\forall i,\ q_i^{\mathsf{max}} = 0$), and there is no queueing delay ($\forall t \geq 0,\ \mathscr{L}(t) = 0$) since service and the arrival rates are the same at all nodes. Hence, the lower bound to the cost is

$$J^{\mathsf{lb}} = \sum_{i=1}^{n} j_i^{\mathsf{c}} \frac{r}{\bar{s}_i}. \tag{5.8}$$

Such a lower bound will be used to compare the quality of the solution later on.

5.2 Controlling the virtual machines

The first node in the chain will see traffic arriving to it at a constant rate of $r(t) = r$. For any node i in the chain, in order to process this traffic entering at this rate, it will need to have a number of

$$\bar{m}_i = \left\lfloor \frac{r}{\bar{s}_i} \right\rfloor, \tag{5.9}$$

machines to always be active. To match the incoming rate r, in addition to the $\bar{m}_i$ machines always on, an additional machine must be on for some time in order to process a request rate of $\bar{s}_i\rho_i$ where ρ_i is the *normalized residual request* rate:

$$\rho_i = \frac{r}{\bar{s}_i} - \bar{m}_i, \qquad \rho_i \in [0,1). \tag{5.10}$$

Naturally, every node can decide arbitrarily when, and for how long, its additional machine is on as long as the average service rate for the nodes matches the incoming request rate r, and the queue does not remain zero for a prolonged amount of time.

With the incoming traffic rate being constant, the first node could achieve this by periodically turning its additional machine on/off. Assuming that this period is T_1, this implies that during every period the additional machine has to process the *residual work* of $T_1 \cdot (r - \bar{m}_1 \bar{s}_1)$. The necessary *on-time* T_1^{on}, needed by the extra machine, to process this work during the period is $T_1^{\mathsf{on}} = T_1 \cdot (r - \bar{m}_1 \bar{s}_1)/\bar{s}_1 = T_1 \rho_1$. The remaining time of the period the additional machine should be switched off, denoted by T_1^{off}, leading to:

$$T_1^{\mathsf{on}} = T\rho_1, \quad T_1^{\mathsf{off}} = T - T_1^{\mathsf{on}} = T \cdot (1 - \rho_1).$$

The second node can also ensure that the average processing capacity of the node matches the incoming traffic rate of r, by periodically turning its additional machine on/off. However, it is slightly more complicated due to the periodic schedule of the first node. The second node could potentially have the additional machine on at a time where the first node has its additional machine off, leading to a scenario where the second node does not utilize all of its available processing capacity.

With the same reasoning, any node i in the chain can ensure that it has an average processing capacity of r by periodically turning an additional machine on/off with a period of T_i. However, finding a suitable schedule for a node i becomes more and more complicated the further down the chain the node is. In fact, finding a suitable schedule increases exponentially with the number of nodes in the chain.

Therefore, to reduce the complexity and make the analysis tractable the extra machines are restricted to be turned on/off with a global period T, i.e. the period by which every node switches on their extra machine is $T_i = T$, leading to:

$$T_i^{\mathsf{on}} = T\rho_i, \quad T_i^{\mathsf{off}} = T - T_i^{\mathsf{on}} = T \cdot (1 - \rho_i). \tag{5.11}$$

Notice, however, that the actual time the extra machine is consuming power is $T_i^{\mathsf{on}} + \Delta_i$ due to the time-overhead for starting a new machine.

The design variable of the optimization problem (5.7) is now the period T, so it remains to investigate how it affects the computing-cost, the buffer-cost, and the end-to-end deadline.

The computing-cost is straightforward to find when the additional machines are switched on/off with a period T. If $\bar{m}_i + 1$ machines are on for a time T_i^{on}, and only $\bar{m}_i$ machines are on for a time T_i^{off}, the cost J_i^{c} of (5.5) becomes:

$$J_i^{\mathsf{c}}(T) = j_i^{\mathsf{c}} \cdot \left(\frac{T_i^{\mathsf{on}} + \Delta_i}{T} + \bar{m}_i \right) = j_i^{\mathsf{c}} \cdot \left(\bar{m}_i + \rho_i + \frac{\Delta_i}{T} \right) \tag{5.12}$$

as long as $T_i^{\mathsf{off}} \geq \Delta_i$. If instead $T_i^{\mathsf{off}} < \Delta_i$, that is if

$$T < \overline{T} := \frac{\Delta_i}{1 - \rho_i}, \tag{5.13}$$

there is no time to switch the additional machine off and then on again before the new period start. Hence, we keep the last machine on, even if it is not processing packets, and the computing cost depends on the period according to:

$$J_i^{\mathsf{c}}(T) = j_i^{\mathsf{c}} \cdot \left(\bar{m}_i + \rho_i + \frac{T_i^{\mathsf{off}}}{T}\right) = j_i^{\mathsf{c}} \cdot (\bar{m}_i + 1). \tag{5.14}$$

It then remains to find the relationship between the period and the maximum queue-length—which by Equation (5.6) translates to the buffer-cost—of the nodes as well as to the maximum end-to-end delay. In Lemma 5.1 below, it is shown that the maximum queue-length is in fact proportional to the period, and similarly, in Lemma 5.2 it is shown that the maximum end-to-end delay is also proportional to the period. The intuition behind this fact is that the longer the period T is, the longer a node will have to wait for the additional machine being off, before turning it on again. During this interval of time, the node is accumulating work and consequently the maximum queue-size is growing leading to the delay for passing through that node growing as well.

LEMMA 5.1
With a constant input rate $r(t) = r$, along with all nodes switching on/off their additional machine with a common period T, the maximum queue size q_i^{max} in the i-th node is

$$q_i^{\mathsf{max}} = T \cdot \theta_i, \tag{5.15}$$

where

$$\begin{aligned}\theta_i = \max\big\{&\rho_i \cdot \big(\bar{s}_i \cdot (1-\rho_i) - \bar{s}_{i-1} \cdot (1-\rho_{i-1})\big),\\ &(1-\rho_{i-1})(\bar{s}_{i-1}\rho_{i-1} - \bar{s}_i\rho_i),\\ &\rho_{i-1} \cdot \big(\bar{s}_{i-1} \cdot (1-\rho_{i-1}) - \bar{s}_i \cdot (1-\rho_i)\big),\\ &(1-\rho_i)(\bar{s}_i\rho_i - \bar{s}_{i-1}\rho_{i-1})\big\},\end{aligned}$$

with ρ_i as defined in (5.10), and T being the period of the switching scheme, common to all nodes. □

Proof The queue size over time $q_i(t)$ is a continuous, piecewise-linear function, since both the input and the service rates are piecewise constant, and the queue size is defined by Equation (5.2). Hence, if at t^* the function $q_i(t)$ takes its maximum value, it must necessarily happen that $\partial q_i(t)/\partial t \geq 0$ in a left-neighborhood of t^* and $\partial q_i(t)/\partial t \leq 0$ in a right-neighborhood of t^*.

To find the value of $\partial q_i(t)/\partial t$, one needs to distinguish among the four possible cases, Case (1a), Case (1b), Case (2a), and Case (2b), depending on the nominal speeds $\bar{s}_{i-1}$ and $\bar{s}_i$, as is shown in Table 5.1. These cases, in turn, determine the sign of $\partial q_i(t)/\partial t$, as summarized in Table 5.2. Note that for $i = 1$, one should consider the preceding node as $(i-1) = 0$, with $\bar{s}_0 = r$, leading to $\bar{m}_0 = 1$ and $\rho_0 = 0$, which would then belong to Case (2b).

Next, the maximum queue-size q_i^{max} will be derived for each case. We will also derive the best time for each node to start its additional machine, i.e. t_i^{on}.

Case (1a)	$(\bar{m}_i+1)\bar{s}_i \geq (\bar{m}_{i-1}+1)\bar{s}_{i-1}$	$\bar{m}_i\bar{s}_i \geq \bar{m}_{i-1}\bar{s}_{i-1}$
Case (1b)	$(\bar{m}_i+1)\bar{s}_i < (\bar{m}_{i-1}+1)\bar{s}_{i-1}$	
Case (2a)	$(\bar{m}_i+1)\bar{s}_i \geq (\bar{m}_{i-1}+1)\bar{s}_{i-1}$	$\bar{m}_i\bar{s}_i < \bar{m}_{i-1}\bar{s}_{i-1}$
Case (2b)	$(\bar{m}_i+1)\bar{s}_i < (\bar{m}_{i-1}+1)\bar{s}_{i-1}$	

Table 5.1 The four possible cases that one needs to distinguish among. Each case is a function of the nominal speeds $\bar{s}_i$ and $\bar{s}_{i-1}$.

$m_{i-1}(t)$	$\bar{m}_{i-1}$	$\bar{m}_{i-1}+1$	$\bar{m}_{i-1}+1$	$\bar{m}_{i-1}$
$m_i(t)$	$\bar{m}_i$	$\bar{m}_i$	$\bar{m}_i+1$	$\bar{m}_i+1$
Case (1a)	≤ 0	≥ 0	≤ 0	≤ 0
Case (1b)	≤ 0	≥ 0	> 0	≤ 0
Case (2a)	> 0	≥ 0	≤ 0	≤ 0
Case (2b)	> 0	≥ 0	> 0	≤ 0

Table 5.2 Sign of $\partial q_i(t)/\partial t$ as function of the number of on-machines within nodes $(i-1)$ and i.

Case (1a) For this case, illustrated in Figure 5.2, the sign of $\partial q_i(t)/\partial t$ shown in Table 5.2, implies that $q_i(t)$ grows only when $m_i(t) = \bar{m}_i$ and $m_{i-1}(t) = \bar{m}_{i-1}+1$. From this condition, the i-th queue can start to decrease either when $m_i(t) \to \bar{m}_i + 1$ or $m_{i-1}(t) \to \bar{m}_{i-1}$. In the first case, the rate of decrease is

$$\begin{aligned} -\partial q_i(t)/\partial t &= \big((\bar{m}_i+1)\cdot\bar{s}_i - (\bar{m}_{i-1}+1)\cdot\bar{s}_{i-1}\big) \\ &= \big(\bar{s}_i\cdot(1-\rho_i) - \bar{s}_{i-1}\cdot(1-\rho_{i-1})\big), \end{aligned}$$

and such a state lasts for T_i^{on} (during the interval of length T_i^{on} in Figure 5.2). This therefore yields a local maximum of:

$$q_i(t_i^{\mathsf{on}}) = T\rho_i\cdot\big(\bar{s}_i\cdot(1-\rho_i) - \bar{s}_{i-1}\cdot(1-\rho_{i-1})\big). \tag{5.16}$$

It is easy to verify that changing the on-time t_i^{on} to instead be later will yield a larger local maximum, and changing it to instead be earlier will yield a negative queue size. The given t_i^{on} is thus the optimal one, and can be expressed relative to t_{i-1}^{on} as:

$$t_i^{\mathsf{on}} = t_{i-1}^{\mathsf{on}} + T\rho_i\frac{\bar{s}_i\cdot(1-\rho_i) - \bar{s}_{i-1}\cdot(1-\rho_{i-1})}{\bar{s}_{i-1}(1-\rho_{i-1}) + \bar{s}_i\rho_i} \tag{5.17}$$

On the other hand, the local maximum when $m_{i-1}(t) \to \bar{m}_{i-1}$ is determined by the interval of length T_{i-1}^{off}, as shown in Figure 5.2, that is

$$T_{i-1}^{\mathsf{off}}\cdot(\bar{m}_i\bar{s}_i - \bar{m}_{i-1}\bar{s}_{i-1}) = T\cdot(1-\rho_{i-1})\cdot(\bar{s}_{i-1}\rho_{i-1} - \bar{s}_i\rho_i).$$

By taking the maximum of the two local maxima, we find

$$q_i^{\mathsf{max}} = T\cdot\max\big\{\rho_i\cdot\big(\bar{s}_i\cdot(1-\rho_i) - \bar{s}_{i-1}\cdot(1-\rho_{i-1})\big), (1-\rho_{i-1})(\bar{s}_{i-1}\rho_{i-1} - \bar{s}_i\rho_i)\big\}.$$

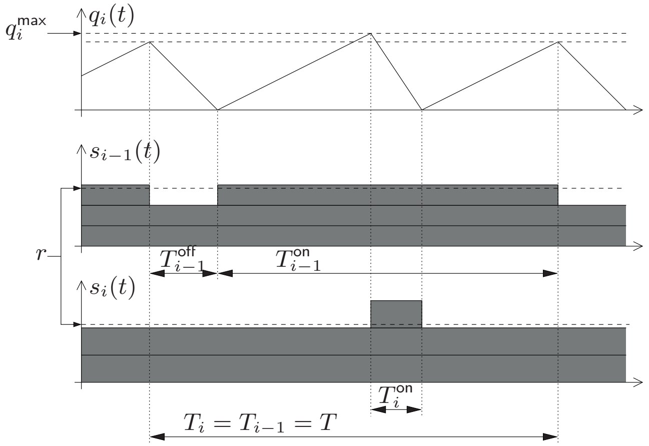


Figure 5.2 Case (1a): service schedule and queue $q_i(t)$. In this example: $r = 17$, $\bar{s}_{i-1} = 6$, $\bar{s}_i = 8$, $T = 120$, $T_{i-1}^{\mathsf{on}} = 100$, $T_i^{\mathsf{on}} = 15$, $q_i^{\mathsf{max}} = 90$.

Case (1b) As shown in Table 5.2, the queue size $q_i(t)$ grows if and only if $\bar{m}_{i-1}+1$ machines are running within the $(i-1)$-th node. The maximum queue size, then, is attained at the instant when such a machine is switched off. To analyze this case, we distinguish between two cases: $T_i^{\mathsf{on}} \geq T_{i-1}^{\mathsf{on}}$ (illustrated in Figure 5.3) and $T_i^{\mathsf{on}} < T_{i-1}^{\mathsf{on}}$ (Figure 5.4). In both cases, to minimize q_i^{max}, the i-th node must start the extra machine simultaneously as the $(i-1)$-th node starts its additional machine in order to reduce the rate of growth of the i-th queue, i.e.

$$t_i^{\mathsf{on}} = t_{i-1}^{\mathsf{on}}. \tag{5.18}$$

Note that the queue size for the i-th node will therefore be zero when it switches on the additional machine,

$$q_i(t_i^{\mathsf{on}}) = 0.$$

To compute q_i^{max}, we examine both when $T_i^{\mathsf{on}} \geq T_{i-1}^{\mathsf{off}}$ (illustrated in Figure 5.3), as well as when $T_i^{\mathsf{on}} < T_{i-1}^{\mathsf{off}}$ (illustrated in Figure 5.4). By considering them both together, we find

$$q_i^{\mathsf{max}} = \max\left\{T_{i-1}^{\mathsf{on}} \cdot \left(\bar{s}_{i-1} \cdot (1-\rho_{i-1}) - \bar{s}_i \cdot (1-\rho_i)\right),\ T_{i-1}^{\mathsf{off}} \cdot (\bar{s}_{i-1}\rho_{i-1} - \bar{s}_i\rho_i)\right\}$$

and, by considering the expressions of T_{i-1}^{on} and T_{i-1}^{off} of Equation (5.11) it can be written as:

$$q_i^{\mathsf{max}} = T \cdot \max\left\{\rho_{i-1} \cdot \left(\bar{s}_{i-1} \cdot (1-\rho_{i-1}) - \bar{s}_i \cdot (1-\rho_i)\right),\ (1-\rho_{i-1})(\bar{s}_{i-1}\rho_{i-1} - \bar{s}_i\rho_i)\right\}.$$

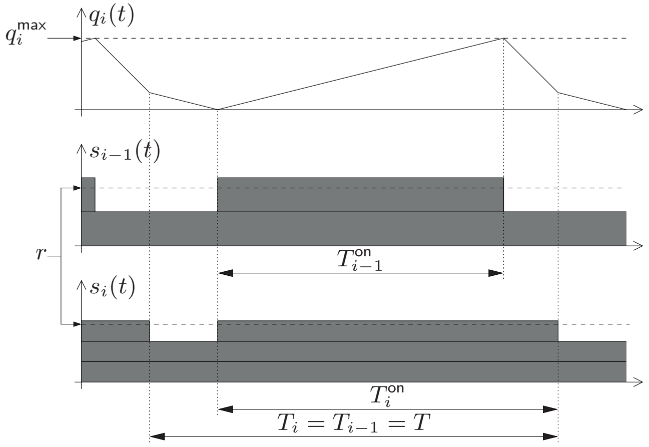


Figure 5.3 Case (1b), $T_i^{\mathsf{on}} \geq T_{i-1}^{\mathsf{on}}$. In this example: $r = 17$, $\bar{s}_{i-1} = 10$, $\bar{s}_i = 6$, $T = 120$, $T_{i-1}^{\mathsf{on}} = 84$, $T_i^{\mathsf{on}} = 100$, $q_i^{\mathsf{max}} = 168$.

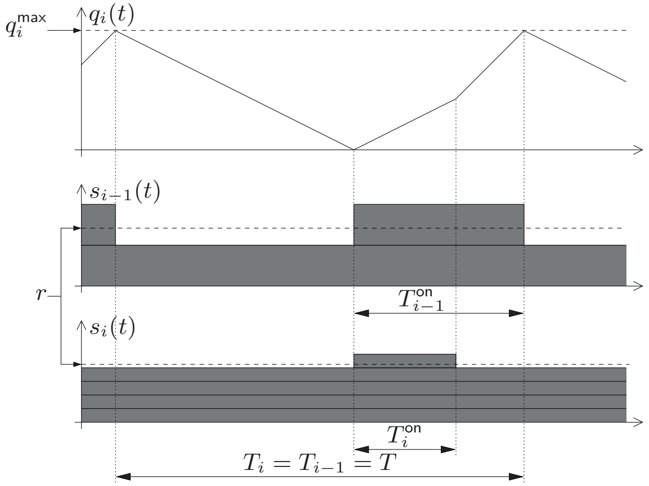


Figure 5.4 Case (1b), $T_i^{\mathsf{on}} < T_{i-1}^{\mathsf{on}}$. In this example: $r = 17$, $\bar{s}_{i-1} = 12$, $\bar{s}_i = 4$, $T = 120$, $T_{i-1}^{\mathsf{on}} = 50$, $T_i^{\mathsf{on}} = 30$, $q_i^{\mathsf{max}} = 280$.

Case (2a) This case is essentially the same as Case (1b). As shown by Table 5.2, the only difference is that $q_i(t)$ is reduced whenever the i-th node has its extra machine on, and grows whenever it is off. This then implies that the maximum queue size is attained when the i-th node switches on the extra machine. To minimize q_i^{max}, the queue size of node i should therefore be empty when it switches off the additional machine. Note that this corresponds to both the i-th node and the $(i-1)$-th node switching off their additional machine simultaneously (compare with Case (1b) where the two nodes switches on their additional machine simultaneously). The time when node i should switch on its additional machine is therefore:

$$t_i^{\mathsf{on}} = \underbrace{t_{i-1}^{\mathsf{on}} + T_{i-1}^{\mathsf{on}}}_{t_{i-1}^{\mathsf{off}}=t_i^{\mathsf{off}}} - T_i^{\mathsf{on}} = t_{i-1}^{\mathsf{on}} + T\cdot(\rho_{i-1}-\rho_i). \tag{5.19}$$

Note that for this case have to consider both $T_i^{\mathsf{on}} \geq T_{i-1}^{\mathsf{on}}$ and $T_i^{\mathsf{on}} < T_{i-1}^{\mathsf{on}}$ when computing $q_i(t_i^{\mathsf{on}})$:

$$q_i(t_i^{\mathsf{on}}) \;=\; \begin{cases} T\cdot(1-\rho_i)(\bar{s}_i\rho_i - \bar{s}_{i-1}\rho_{i-1}), & T_i^{\mathsf{on}} \geq T_{i-1}^{\mathsf{on}} \\ T\rho_i\cdot\big(\bar{s}_i\cdot(1-\rho_i) - \bar{s}_{i-1}\cdot(1-\rho_{i-1})\big), & T_i^{\mathsf{on}} < T_{i-1}^{\mathsf{on}} \end{cases} \tag{5.20}$$

The maximum queue size is, as stated earlier, found when F_i switches on its extra machine. By considering $T_i^{\mathsf{on}} \geq T_{i-1}^{\mathsf{on}}$ and $T_i^{\mathsf{on}} < T_{i-1}^{\mathsf{on}}$ together, the expression for q_i^{max} can be combined into:

$$q_i^{\mathsf{max}} = T\cdot\max\left\{\rho_i\cdot\big(\bar{s}_i\cdot(1-\rho_i)-\bar{s}_{i-1}(1-\rho_{i-1})\big),(1-\rho_i)(\bar{s}_i\rho_i-\bar{s}_{i-1}\rho_{i-1})\right\}$$

Case (2b) Table 5.2 shows the similarity between this case and Case (1a), with the difference being that for this case $q_i(t)$ only shrinks when $m_i(t) = \bar{m}_i + 1$ and $m_{i-1}(t) = \bar{m}_{i-1}$. Therefore, $q_i(t)$ will always grow when $m_{i-1}(t) = \bar{m}_{i-1}+1$. To reduce the rate of this growth the i-th node should therefore have its extra machine on whenever the $(i-1)$-th node has its extra machine on. Furthermore, to reduce the local maximum attained at the end of this growth, the i-th node should also switch on its additional machine such that $q_i(t)$ is empty when it does so, e.g., $q_i(t_{i-1}^{\mathsf{on}}) = 0$. Furthermore, since $q_i(t)$ grows when both the i-th node and the $(i-1)$-th node has its additional machine off, there is also a local maximum of $q_i(t)$ attained when node i switches on its additional machine. To minimize this local maximum, the i-th node should ensure that $q_i(t)$ is empty when it switches off its additional machine, i.e. $q_i(t_i^{\mathsf{off}}) = 0$. The on-switching time should thus be:

$$t_i^{\mathsf{on}} = t_{i-1}^{\mathsf{on}} - \frac{q_i(t_i^{\mathsf{on}})}{\bar{s}_i\cdot(1-\rho_i)+\bar{s}_{i-1}\rho_{i-1}} \tag{5.21}$$

where

$$\begin{aligned} q_i(t_i^{\mathsf{on}}) &= T_i^{\mathsf{off}}\cdot(\bar{m}_{i-1}\bar{s}_{i-1} - \bar{m}_i\bar{s}_i) \\ &= T\cdot(1-\rho_i)(\bar{s}_i\rho_i - \bar{s}_{i-1}\rho_{i-1}). \end{aligned} \tag{5.22}$$

The other local maximum, occurring when the $(i-1)$-th node switches off its additional machine is therefore:

$$\begin{aligned} q_i(t_{i-1}^{\mathsf{off}}) &= T_{i-1}^{\mathsf{on}}\cdot\big((\bar{m}_{i-1}+1)\cdot\bar{s}_{i-1} - (\bar{m}_i+1)\cdot\bar{s}_i\big) \\ &= T\rho_{i-1}\cdot\big(\bar{s}_{i-1}\cdot(1-\rho_{i-1}) - \bar{s}_i\cdot(1-\rho_i)\big) \end{aligned}$$

The maximum queue-size for this case thus given by:

$$q_i^{\mathsf{max}} = T \cdot \max \left\{ \rho_{i-1} \cdot \left(\bar{s}_{i-1} \cdot (1-\rho_{i-1}) - \bar{s}_i \cdot (1-\rho_i) \right),\ (1-\rho_i)(\bar{s}_i \rho_i - \bar{s}_{i-1}\rho_{i-1}) \right\}.$$

Conclusion By taking the maximum among all four cases, Equation (5.15) is found and the Lemma is proved. □

The expression for q_i^{max} in (5.15) suggests that the **maximum queue-length is always bounded** with respect to the input rate to the service-chain, as shown in Corollary 5.1 below. The intuition behind this is that regardless the size of the input rate, it is possible to find a number $\bar{m}_i$ such that $\bar{s}_i \cdot (\bar{m}_i + 1) > r$, hence one can always match the input rate.

COROLLARY 5.1
The maximum queue size q_i^{max} of any node i is always bounded, regardless of the rate r of the input.

Proof From the definition of ρ_i in Equation (5.10), it always holds that $\rho_i \in [0,1)$. Hence, from the expression of (5.15), it follows that q_i^{max} is always bounded. □

Finally, in order to solve the optimal design problem one has to relate the period T and the largest end-to-end latency:

LEMMA 5.2
With a constant input rate, $r(t) = r$, the largest end-to-end latency $\mathscr{L}(t)$ for any request passing through nodes 1 thru n is

$$\forall t \geq 0, \qquad \mathscr{L}(t) \leq T \cdot \sum_{i=1}^{n} \gamma_i. \tag{5.23}$$

where γ_i depends on r, $\bar{s}_i$, and $\bar{s}_{i-1}$ given in Table 5.3, with the four different cases given in Table 5.1. □

Proof With a constant input $r(t) = r$ to the service chain, the maximum end-to-end latency up until node i, i.e., $\mathscr{L}_i(t)$, is bounded by

$$\forall t \geq 0, \quad \mathscr{L}_i(t) \leq \max_{t \geq 0} \left\{ \frac{R_0(t) - S_i(t)}{r} \right\} = \max_{t \geq 0} \left\{ t - \frac{S_i(t)}{r} \right\}, \tag{5.24}$$

with $S_i(t)$ being the cumulative served request by node i, as in Equation (3.1), and $R_0(t) = \int_0^t r \mathrm{d}x$ is the cumulative arrived requests to the chain. Since $S_i(t)$ is a piece-wise linear function, growing with rates $\bar{s}_i \cdot \bar{m}_i$ or $\bar{s}_i \cdot (\bar{m}_i + 1)$, it follows that the maximum end-to-end latency up to the i-th node, $\mathscr{L}_i(t)$, is attained when the i-th node switches on the additional machine (denoted by t_i^{on}), that is

$$\max_{t \geq 0} \mathscr{L}_i(t) = \max_{t} \left\{ t - \frac{S_i(t)}{r} \right\} = t_i^{\mathsf{on}} - \frac{S_i(t_i^{\mathsf{on}})}{r}. \tag{5.25}$$

Case	γ_i
Case (1a)	$\frac{1}{r}\bar{s}_i\rho_i^2\frac{\bar{s}_i\cdot(1-\rho_i)-\bar{s}_{i-1}\cdot(1-\rho_{i-1})}{\bar{s}_{i-1}\cdot(1-\rho_{i-1})+\bar{s}_i\rho_i}$
Case (1b)	0
Case (2a)	$\begin{cases} \frac{1}{r}\big(\bar{s}_{i-1}\rho_{i-1}\cdot(\rho_{i-1}-\rho_i)+ \\ \quad +(1-\rho_i)(\bar{s}_i\rho_i-\bar{s}_{i-1}\rho_{i-1}\big), & T_i^{\text{on}}\geq T_{i-1}^{\text{on}} \\ \frac{1}{r}\Big(\rho_i\cdot\big(\bar{s}_i\cdot(1-\rho_i)-\bar{s}_{i-1}\cdot(1-\rho_{i-1})\big)+ \\ \quad +\bar{s}_{i-1}\cdot(\rho_{i-1}-1)(\rho_{i-1}-\rho_i)\Big), & T_i^{\text{on}}<T_{i-1}^{\text{on}} \end{cases}$
Case (2b)	$\frac{1}{r}\bar{s}_i\cdot(1-\rho_i)^2\frac{\bar{s}_i\rho_i-\bar{s}_{i-1}\rho_{i-1}}{\bar{s}_i\cdot(1-\rho_i)+\bar{s}_{i-1}\rho_{i-1}}$

Table 5.3 The γ_i given for each of the four cases (presented in Table 5.1).

Adding another node after the i-th node, will thus add L_{i+1}^+ time-units to the maximum end-to-end latency. Note that L_{i+1}^+ might be zero. It is therefore possible to write the maximum end-to-end latency $\mathscr{L}_i(t)$ as:

$$\max_{t\geq 0}\mathscr{L}_i(t)=\max_{t\geq 0}\mathscr{L}_{i-1}(t)+L_i^+. \tag{5.26}$$

Equation (5.25) then implies that

$$\begin{aligned}
D_i^* &= \max_{t\geq 0}\{\mathscr{L}_i(t)\}-\max_{t\geq 0}\{\mathscr{L}_{i-1}(t)\} \\
&= t_i^{\text{on}}-\frac{S_i(t_i^{\text{on}})}{r}-t_{i-1}^{\text{on}}+\frac{S_{i-1}(t_{i-1}^{\text{on}})}{r} \\
&= t_i^{\text{on}}-t_{i-1}^{\text{on}}+\frac{S_{i-1}(t_{i-1}^{\text{on}})-S_i(t_i^{\text{on}})}{r} \\
&= t_i^{\text{on}}-t_{i-1}^{\text{on}}+\frac{\overbrace{S_{i-1}(t_i^{\text{on}})-S_i(t_i^{\text{on}})}^{q_i(t_i^{\text{on}})}+\int_{t_i^{\text{on}}}^{t_{i-1}^{\text{on}}}s_{i-1}(x)dx}{r} \\
&= t_i^{\text{on}}-t_{i-1}^{\text{on}}+\frac{q_i(t_i^{\text{on}})}{r}+\frac{1}{r}\int_{t_i^{\text{on}}}^{t_{i-1}^{\text{on}}}s_{i-1}(x)dx \\
&= t_i^{\text{on}}-t_{i-1}^{\text{on}}+\frac{q_i(t_i^{\text{on}})}{r}+(t_{i-1}^{\text{on}}-t_i^{\text{on}})\frac{s_{i-1}^*}{r} \\
&= \frac{q_i(t_i^{\text{on}})}{r}+(t_i^{\text{on}}-t_{i-1}^{\text{on}})(1-\frac{s_{i-1}^*}{r}),
\end{aligned}$$

where $\int_{t_i^{\text{on}}}^{t_{i-1}^{\text{on}}}s_{i-1}(x)dx=(t_{i-1}^{\text{on}}-t_i^{\text{on}})\cdot s_{i-1}^*$ since $s_{i-1}(t)$ is a piecewise constant function, changing value only in t_{i-1}^{on}. The values of s_{i-1}^* depend on whether the $(i-1)$-th node has its additional machine on or off during this time-interval. It should also be noted that when $t_i^{\text{on}}\geq t_{i-1}^{\text{on}}$, the $(i-1)$-th node will start its additional machine before node i, implying that the $(i-1)$-th node will have $(\bar{m}_{i-1}+1)$ machines on during the time-interval $[t_{i-1}^{\mathsf{on}}, t_i^{\mathsf{on}}]$. On the other hand, if $t_i^{\mathsf{on}} < t_{i-1}^{\mathsf{on}}$, it follows that the i-th node will start its additional machine before the $(i-1)$-th node does so. In that case the $(i-1)$-th node will only have $\bar{m}_{i-1}$ machines on during the time-interval $[t_{i-1}^{\mathsf{on}}, t_i^{\mathsf{on}}]$. Hence, s_{i-1}^* can be written as:

$$s_{i-1}^* = \begin{cases} \bar{s}_{i-1} \cdot (\bar{m}_{i-1} + 1), & t_i^{\mathsf{on}} \geq t_{i-1}^{\mathsf{on}} \\ \bar{s}_{i-1} \cdot \bar{m}_{i-1}, & t_i^{\mathsf{on}} < t_{i-1}^{\mathsf{on}} \end{cases}.$$

It should also be noted that t_i^{on} and t_{i-1}^{on} are such that the time between them is the smallest possible. Hence, $(t_i^{\mathsf{on}} - t_{i-1}^{\mathsf{on}})$ might be positive or negative, and corresponds to the expressions derived in Lemma 5.1, Equations (5.17), (5.18), (5.19), and (5.21) for Case (1a)–Case (2b) respectively.

When $t_i^{\mathsf{on}} \geq t_{i-1}^{\mathsf{on}}$ we can therefore write L_i^+ as

$$\begin{aligned} L_i^+ &= (t_i^{\mathsf{on}} - t_{i-1}^{\mathsf{on}})\Big(1 - \tfrac{\bar{s}_{i-1}(\bar{m}_{i-1}+1)}{r}\Big) + \frac{q_i(t_i^{\mathsf{on}})}{r} \\ &= \frac{\bar{s}_{i-1}}{r} \cdot (t_i^{\mathsf{on}} - t_{i-1}^{\mathsf{on}})\Big(\underbrace{\frac{r}{\bar{s}_{i-1}} - \bar{m}_i}_{=\rho_{i-1}} - 1\Big) + \frac{q_i(t_i^{\mathsf{on}})}{r} \\ &= \frac{\bar{s}_{i-1}}{r} \cdot (t_i^{\mathsf{on}} - t_{i-1}^{\mathsf{on}})(\rho_{i-1} - 1) + \frac{q_i(t_i^{\mathsf{on}})}{r}. \end{aligned} \tag{5.27}$$

In the opposite case, when $t_i^{\mathsf{on}} < t_{i-1}^{\mathsf{on}}$ we instead get

$$\begin{aligned} L_i^+ &= (t_i^{\mathsf{on}} - t_{i-1}^{\mathsf{on}})\Big(1 - \frac{\bar{s}_{i-1}\bar{m}_{i-1}}{r}\Big) + \frac{q_i(t_i^{\mathsf{on}})}{r} \\ &= \frac{\bar{s}_{i-1}}{r} \cdot (t_i^{\mathsf{on}} - t_{i-1}^{\mathsf{on}})\Big(\frac{r}{\bar{s}_{i-1}} - \bar{m}_i\Big) + \frac{q_i(t_i^{\mathsf{on}})}{r} \\ &= \frac{\bar{s}_{i-1}}{r} \cdot (t_i^{\mathsf{on}} - t_{i-1}^{\mathsf{on}}) \cdot \rho_{i-1} + \frac{q_i(t_i^{\mathsf{on}})}{r}. \end{aligned}$$

In Lemma 5.1, both $(t_i^{\mathsf{on}} - t_{i-1}^{\mathsf{on}})$ and $q_i(t_i^{\mathsf{on}})$ were derived for Cases (1a)–(2b) in Equations (5.16)–(5.22). Therefore, we can derive, L_i^+ for each of the four cases as:

Case (1a) For this case, it always holds that $t_i^{\mathsf{on}} \geq t_{i-1}^{\mathsf{on}}$. Hence by inserting $q_i(t_i^{\mathsf{on}})$ of Equation (5.16) and $(t_i^{\mathsf{on}} - t_{i-1}^{\mathsf{on}})$ of Equation (5.17) into Equation (5.27) we can write L_i^+ as

$$L_i^+ = T \cdot \frac{1}{r}\bar{s}_i\rho_i^2 \frac{\bar{s}_i \cdot (1-\rho_i) - \bar{s}_{i-1} \cdot (1-\rho_{i-1})}{\bar{s}_{i-1} \cdot (1-\rho_{i-1}) + \bar{s}_i\rho_i}.$$

Case (1b) For this case Equation (5.18) imply that $t_i^{\mathsf{on}} = t_{i-1}^{\mathsf{on}}$ and that $q_i(t_i^{\mathsf{on}})$ is always 0. Hence, L_i^+ will always be 0, implying that $\gamma_i = 0$.

Case (2a) Here one must distinguish between two cases: $T_i^{\mathsf{on}} \geq T_{i-1}^{\mathsf{on}}$ and $T_i^{\mathsf{on}} < T_{i-1}^{\mathsf{on}}$. When $T_i^{\mathsf{on}} \geq T_{i-1}^{\mathsf{on}}$ it always hold that $t_i^{\mathsf{on}} \leq t_{i-1}^{\mathsf{on}}$. Hence, by inserting $(t_i^{\mathsf{on}} - t_{i-1}^{\mathsf{on}})$ and $q_i(t_i^{\mathsf{on}})$ given by Equations (5.19)–(5.20) into Equation (5.28) we can write L_i^+ as

$$L_i^+ = T\frac{1}{r}\big(\bar{s}_{i-1}\rho_{i-1}\cdot(\rho_{i-1}-\rho_i) + (1-\rho_i)(\bar{s}_i\rho_i - \bar{s}_{i-1}\rho_{i-1}\big).$$

When $T_i^{\mathsf{on}} < T_{i-1}^{\mathsf{on}}$, it instead holds that $t_i^{\mathsf{on}} \geq t_{i-1}^{\mathsf{on}}$. Therefore, by inserting Equations (5.19)–(5.20) into Equation (5.27) we can write L_i^+ as

$$L_i^+ = T\frac{1}{r}\Big(\rho_i\cdot\big(\bar{s}_i\cdot(1-\rho_i) - \bar{s}_{i-1}\cdot(1-\rho_{i-1})\big) + \bar{s}_{i-1}\cdot(\rho_{i-1}-1)(\rho_{i-1}-\rho_i)\Big).$$

Case (2b) For this case, it always holds that $t_i^{\mathsf{on}} \leq t_{i-1}^{\mathsf{on}}$. Therefore, by inserting $(t_i^{\mathsf{on}} - t_{i-1}^{\mathsf{on}})$ and $q_i(t_i^{\mathsf{on}})$ given by Equation (5.21)–(5.22) into Equation (5.28) we can write L_i^+ as

$$L_i^+ = T\frac{1}{r}\bar{s}_i\cdot(1-\rho_i)^2\frac{\bar{s}_i\rho_i - \bar{s}_{i-1}\rho_{i-1}}{\bar{s}_i\cdot(1-\rho_i)+\bar{s}_{i-1}\rho_{i-1}}.$$

Conclusion It is now possible to conclude that for all four cases it is one can write $L_i^+ = T\gamma_i$, with γ_i depending only on r, $\bar{s}_i$, and $\bar{s}_{i-1}$. Note that Equations (5.9)–(5.10) imply that ρ_i (and ρ_{i-1}) depend on r and $\bar{s}_i$ (and $\bar{s}_{i-1}$). We can then conclude the proof since Equation (5.26) implies that the maximum queueing delay is

$$\forall t \geq 0, \quad \mathscr{L}(t) \leq \max_{t\geq 0}\mathscr{L}(t) = \sum_{i=1}^{n} L_i^+ = T\cdot\sum_{i=1}^{n}\gamma_i. \qquad \square$$

Solution to the optimization problem

With the relationships between the cost, the end-to-end delay, and the period established, the optimization problem (5.7) can be written as

$$J(T) = T\sum_{i=1}^{n} j_i^{\mathsf{q}}\theta_i + \sum_{i:T<\overline{T}_i} j_i^{\mathsf{c}}\cdot(1-\rho_i) + \sum_{i:T\geq\overline{T}_i} j_i^{\mathsf{c}}\frac{\Delta_i}{T} + J^{\mathsf{lb}}, \tag{5.28}$$

where θ_i is given by Lemma 5.1, J^{lb} is the lower bound given by (5.8) , and $\overline{T}_i$ is the value of the period, below for which it is not feasible to switch the additional machine off and then on again, given by (5.13):

$$T < \overline{T}_i \;\Leftrightarrow\; T_i^{\mathsf{off}} < \Delta_i.$$

In fact, $\forall i$ with $T < \overline{T}_i$ we pay the full cost of having $\bar{m}_i + 1$ machines always on. The deadline constraint in (5.7), can be simply written as

$$T \leq c := \frac{\mathscr{D}}{\sum_{i=1}^{n}\gamma_i},$$

with γ_i given in Lemma 5.2, Table 5.3.

The cost $J(T)$ of (5.28) is a continuous function of one variable T. It has to be minimized over the closed interval $[0, c]$. Hence, by the Weierstraß's extreme-value theorem, it has a minimum. To find this minimum, we just check all (finite) points at which the cost is not differentiable and the ones where the derivative is equal to zero. Let us define all points in $[0, c]$ in which $J(T)$ is not differentiable:

$$\mathscr{C} = \{\overline{T}_i : \overline{T}_i < c\} \cup \{0\} \cup \{c\}. \tag{5.29}$$

We denote by $p = |\mathscr{C}| \leq n+2$ the number of points in $\mathscr{C}$. Also, we denote by $c_k \in \mathscr{C}$ the points in $\mathscr{C}$ and we assume they are ordered increasingly $c_1 < c_2 < \ldots < c_p$. Since the cost $J(T)$ is differentiable over the open interval (c_k, c_{k+1}), the minimum may also occur at an interior point of (c_k, c_{k+1}) with derivative equal to zero. Let us denote by $\mathscr{C}^*$ the set of all interior points of (c_k, c_{k+1}) with derivative of $J(T)$ equal to zero, that is

$$\mathscr{C}^* = \{c_k^* : k = 1, \ldots, p-1,\ c_k < c_k^* < c_{k+1}\} \tag{5.30}$$

with

$$c_k^* = \sqrt{\frac{\sum_{i:\overline{T}_i < c_{k+1}} j_i^{\mathsf{c}} \Delta_i}{\sum_{i=1}^n j_i^{\mathsf{q}} \theta_i}}.$$

Then, the optimal period is given by

$$T^* = \arg \min_{T \in \mathscr{C} \cup \mathscr{C}^*} \{J(T)\}. \tag{5.31}$$

Example – solving the optimization problem

To illustrate how to solve the optimization problem a simple service chain of two nodes is used as an example. The input rate to the chain is $r = 17 \cdot 10^3$ packets per second. Every request has an end-to-end-deadline of $\mathscr{D} = 0.02$ seconds. The parameters of the two nodes are reported in Table 5.4.

i	$\bar{s}_i$ (pps)	j_i^{c}	j_i^{q}	Δ_i (s)
1	$6 \cdot 10^3$	6	$0.5 \cdot 10^{-3}$	0.01
2	$8 \cdot 10^3$	8	$0.5 \cdot 10^{-3}$	0.01

Table 5.4 Parameters of the example.

As mentioned earlier, the input $r(t) = r$ can be seen as "dummy node" $i = 0$ preceding node $i = 1$, with $\bar{s}_0 = r$, $\bar{m}_0 = 1$, and $\rho_0 = 0$. When deriving the schedule, it follows from (5.9) and (5.10) that $\bar{m}_1 = \bar{m}_2 = 2$, and $\rho_1 = \frac{5}{6}$, $\rho_2 = \frac{1}{8}$, implying that both nodes must always keep two machines on, and then periodically switch a third one on/off. This also implies that the two nodes belong to Case (1a), and that

$\overline{T}_1 = 60.0 \cdot 10^{-3}$ and $\overline{T}_2 = 11.4 \cdot 10^{-3}$, where $\overline{T}_i$ is the threshold period for the i-th node, as defined in Equation (5.13).

From Lemma 5.1 it follows that the expression $\sum_{i=1}^{n} j_i^{\mathrm{q}} \theta_i$ in the cost function (5.28) is evaluated to $\sum_{i=1}^{n} j_i^{\mathrm{q}} \theta_i = 0.792$. Moreover from Lemma 5.2 it follows that γ_i determining the extra queuing delay introduced by each node, are $\gamma_1 = 49.0 \cdot 10^{-3}$ and $\gamma_2 = 22.1 \cdot 10^{-3}$, which in turn leads to

$$c = \frac{\mathscr{D}}{\gamma_1 + \gamma_2} = \frac{0.02}{71.1 \cdot 10^{-3}} = 281 \cdot 10^{-3}.$$

Since $\overline{T}_2 < \overline{T}_1 < c$, the set $\mathscr{C}$ of Equation (5.29) containing the boundary is

$$\mathscr{C} = \{0, \underbrace{0.00114}_{\overline{T}_2}, \underbrace{0.060}_{\overline{T}_1}, \underbrace{0.281}_{c}\}.$$

To compute the set $\mathscr{C}^*$ of interior points with derivative equal to zero defined in (5.30), which is needed to compute the period with minimum cost from (5.31), we must check all intervals with boundaries at two consecutive points in $\mathscr{C}$. In the interval $(0, \overline{T}_2)$ the derivative of J is never zero. When checking the interval $(\overline{T}_2, \overline{T}_1)$, the derivative is zero at

$$c_1^* = \sqrt{\frac{j_2^{\mathrm{c}} \Delta_2}{\sum_{i=1}^{n} j_i^{\mathrm{q}} \theta_i}} = 0.318,$$

which, however, falls outside the interval. Finally, when checking the interval $(\overline{T}_1, c)$ the derivative is zero at

$$c_2^* = \sqrt{\frac{j_1^{\mathrm{c}} \Delta_1 + j_2^{\mathrm{c}} \Delta_2}{\sum_{i=1}^{n} j_i^{\mathrm{q}} \theta_i}} = 0.421 > c = 0.281.$$

Hence, the set of points with derivatives equal to zero is $\mathscr{C}^* = \emptyset$. By inspecting the cost at points in $\mathscr{C}$ we find that the minimum occurs at $T^* = c = 0.281$, with cost $J(T^*) = 34.7$.

The period of T^* results in a schedule for turning on/off the additional machine as is illustrated in Figure 5.5. One can see how the first node needs a longer on-time for the additional machine, compared to the second node. One can also see that when the additional machine is off for both nodes, the queue of the second node is reduced. This is very similar to the illustration of Figure 5.2, since indeed the two nodes of this example belong to Case (1a).

The periodic schedule of these two nodes lead to the queue-size of them growing and reducing periodically. This is also illustrated in Figure 5.6 where the trajectory of the queue-sizes is shown in a state-space diagram. One can easily see that this yields a classic limit-cycle.

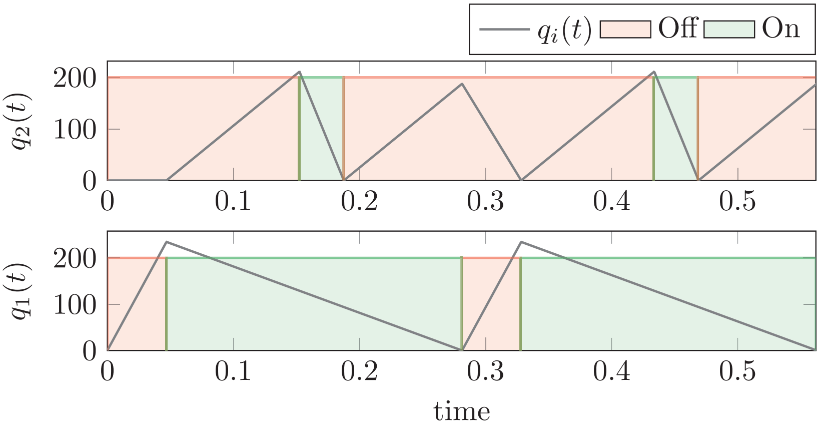


Figure 5.5 Evolution of the queue-size for the two queues in the example, along with an illustration of when their additional machine is turned on/off. One can see the similarities with Figure 5.2, since the two nodes in this example also belong to Case (1a).

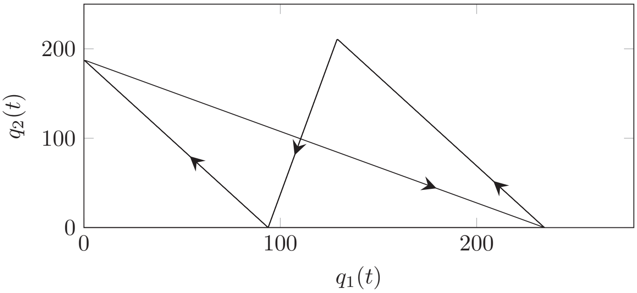


Figure 5.6 Nominal state-space trajectory of the queues for the two nodes in the example. One can see that the periodic schedule of the two nodes, illustrated in Figure 5.5, results in a limit-cycle.

Adding feedback for increased robustness

In this section a feedback-law will be derived to increase the robustness and stability of the system. The goal is to enable the system to handle impulse disturbances of mass d_i occurring at the i-th node, i.e., d_i packets suddenly appear at the tail of the i-th queue. Such a disturbance could be used to model for instance a stochastic input signal, small deviations in the processing capacities of the nodes, or if the nodes are not perfectly synchronized. The feedback will use information about deviations from the expected queue-sizes and use this to dynamically change the on-time of the additional machines in order to drive the system back to the desired queue-sizes and, as illustrated in Figure 5.7 where the nominal schedule is shown in green and the adjusted, extra on-time for the i-th node, highlighted in red. By extending the on-time during the second period the extra work introduced by the impulse-disturbance is processed and the system is driven back to the desired state. What is also highlighted in Figure 5.7 is that one can use an impulse disturbance to model both modeling errors as well as time-varying input. For instance, denoting the expected queue-size by $q_i(t)$ and the true queue-size by $\tilde{q}_i(t)$, one can model the difference of them at time t^* by an impulse disturbance of mass $d_i = \tilde{q}_i(t^*) - q_i(t^*)$. One cause for such deviations might be a stochastic input rate.

When an impulse disturbance of mass d_i appears at the i-th node, the nominal on-time T_i^{on} will not be sufficient to process both the residual work, i.e. the work not processed by the $\bar{m}_i$ machines, and the impulse disturbance. Hence, at the end of T_i^{on} there will still be d_i too many packets in the queue. Moreover, without any feedback law to adjust T_i^{on} the extra load will never be processed.

The time needed by the additional machine to process the impulse disturbance is $d_i/\bar{s}_i$, which should thus be added to the nominal on-time T_i^{on}. Denoting the adjusted on-time by $\tilde{T}_i^{\text{on}}$ it should thus be given by

$$\tilde{T}_i^{\text{on}} = T_i^{\text{on}} + \frac{d_i}{\bar{s}_i}. \tag{5.32}$$

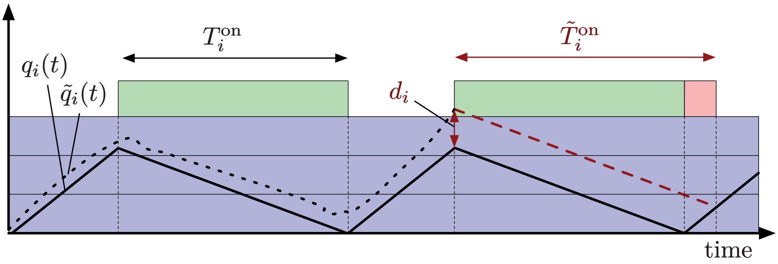


Figure 5.7 Illustration of the idea to alter the on-time of the additional machine in order to handle the extra load caused by the impulse disturbance. The blue bars symbolize the machines that are always on, and the green symbolize when the extra machine is supposed to be on, and the red bar highlight the additional on-time for the extra machine needed to process the extra work introduced by the impulse disturbance. The solid (–) line show the expected queue-size of the node, and the dashed (- - -) line show the true queue-size. One can thus use the impulse disturbance as a tool to model the difference between these when the additional machine starts.

However, since the on-time can only be extended by converting off-time into on-time, it might very well be that $d_i/\bar{s}_i > T_i^{\text{off}}$, implying that there is not sufficient off-time in the next period to convert to on-time in order to process the extra work caused by the impulse disturbance. In fact, assuming that no additional disturbances occur during the processing of d_i, the node will need $\lceil \frac{(d_i/\bar{s}_i)}{T_i^{\text{off}}} \rceil$ periods before the extra work is fully processed. Hence, the total time needed is

$$\tilde{T}_i^{\text{on}} = T \cdot \underbrace{\left\lfloor \frac{d_i/\bar{s}_i}{T_i^{\text{off}}} \right\rfloor}_{\text{number of full periods needed}} + T_i^{\text{on}} + T_i^{\text{off}} \cdot \overbrace{\left(\frac{d_i/\bar{s}_i}{T_i^{\text{off}}} - \left\lfloor \frac{d_i/\bar{s}_i}{T_i^{\text{off}}} \right\rfloor \right)}^{\text{fraction of final } T_i^{\text{off}} \text{ needed } \in [0,1)}.$$

Here one should note that $\tilde{T}_i^{\text{on}} \to \infty$ as $T_i^{\text{off}} \to 0$, therefore, should $\tilde{T}_i^{\text{on}}$ grow very large it might be favorable to switch on yet another machine. If such a thing would happen it would thus need to switch between using $\bar{m}_i + 1$ and $\bar{m}_i + 2$ machines, which is the problem studied in this chapter.

To allow for this impulse disturbance to be accepted by the buffer there must be space in the buffer. Therefore, one would have to increase the maximum queue-size $q_i^{\max}$ given by Lemma 5.1 to

$$\tilde{q}_i^{\max} = q_i^{\max} + d_i.$$

Naturally one might wonder if this affects the solution of the optimization problem (5.7) and the answer is no. In the cost function (5.28), the added queue-size would add a cost

$$\sum_{i=1}^{n} j_i^{\text{q}} d_i$$

which does not depend on the optimization variable T, hence implying a linear shift of the cost and thus the same optimal period T still holds. The same holds for the additional time needed to process the impulse disturbance, implying a larger computation cost.

Designing the schedule

To implement this in the real system, one would have to know when to start the extra machines, i.e. to derive a schedule. With the period T by which the additional machines should be switched on/off, along with the adaptive on-time $\tilde{T}_i^{\text{on}}$ to handle impulse disturbances and variations in the input, the only thing one needs to know before designing a schedule is *when to start the additional machine for the first time*.

With the schedule being periodic with period T, the $(k+1)$-th time the additional machine in the i-th node starts is given by

$$t_{i,k}^{\text{on}} = t_{i,0}^{\text{on}} + kT, \quad i = 1, 2, \ldots, \quad \forall k \geq 1$$

where $t_{i,0}^{\text{on}} \geq 0$ is the first time the additional machine starts in the node. Similarly, the $(k+1)$-th time the additional machine should stop is given by

$$t_{i,k}^{\text{off}} = t_{i,k}^{\text{on}} + \tilde{T}_i^{\text{on}}, \quad i = 1, 2, \ldots, \quad \forall k \geq 1$$

where $\tilde{T}_i^{\text{on}}$ is given by the feedback-law (5.32), but with d_i being the difference between the expected queue size at this time-instance and the actual queue-size, as shown later. Hence, it remains to define t_i^{on} and d_i.

When proving Lemma 5.1 a by-product was the optimal time to start the additional machine as well as the expected queue-size at that time. Here it should be noted that for the first node in the chain, one can regard the input r as the output of a "dummy node" $i = 0$, with $\bar{s}_0 = r$, $\bar{m}_0 = 1$, and $\rho_0 = 0$, leading to the node $i = 1$ belonging to Case (2b). Node $i = 0$ is denoted a "dummy node" since it is not an actual node in the chain, but instead a useful concept as it helps when solving the optimization problem as well as when designing the schedule. Moreover, for this "dummy node", its starting time is assumed to be 0, i.e., $t_0^{\text{on}} = 0$. Finally, the on-time computed when the additional machine starts every period, is given by

$$\tilde{T}_{i,k}^{\text{on}} = \min\left(T,\, T_i^{\text{on}} + \frac{q_i(t_{i,0}^{\text{on}}) - q_i(t_{i,k}^{\text{on}})}{\bar{s}_i}\right),$$

where the *min*-statement ensures that if there is a large difference between the expected actual queue-sizes the additional machine is kept on for a period, and then a new on-time is computed.

5.3 Evaluation of the feedback

This section will illustrate, what happens in the presence of an impulse disturbance as well as when the incoming traffic is not static but stochastic. Finally, a more thorough comparison is made between when feedback is used or not. This evaluation is done by simulating a large number of randomly generated service-chains and then evaluate the probability of a chain missing a deadline.

Example – impulse disturbance

Should there be an impulse disturbance at node i, i.e. that d_i packets suddenly appear at the tail of the queue in the i-th node, the nominal limit-cycle of the queues (shown in Figure 5.6) will be perturbed. This is illustrated in Figure 5.8 where we gave the first node an impulse disturbance of mass d_1 after 1.5 periods. With no feedback-law in place, the nominal limit-cycle is never recovered. Instead it will be perturbed a distance d_i in the q_i-th direction of the state-space diagram.

When extending the system with the feedback-law described in Section 5.2 the nodes will dynamically change the on-time of the additional machines. This leads to the nominal limit-cycle being recovered after the initial impulse disturbance, as

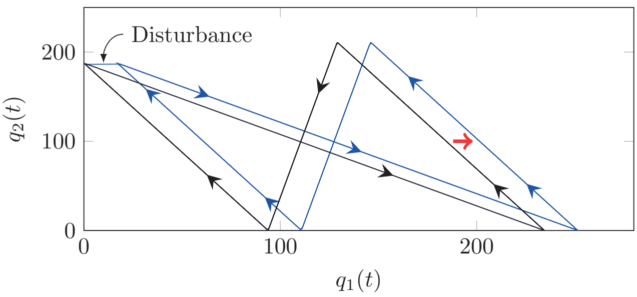


Figure 5.8 When the system is given an impulse disturbance of mass d_1 to the first node, the nominal limit-cycle (black) is perturbed to the right, giving a new limit-cycle (blue). Without a feedback-law, the nominal limit-cycle is never recovered.

shown in Figure 5.9. One can see that the recovery takes around two periods, ending when the blue trajectory hits the black one.

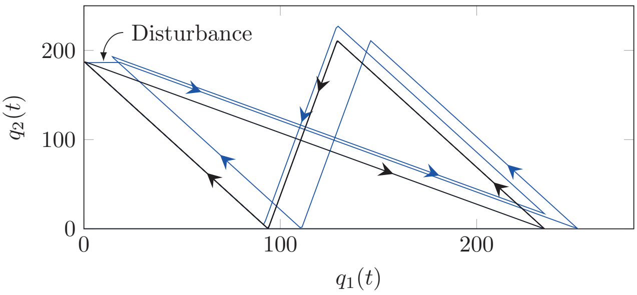


Figure 5.9 Using the feedback-law it successfully recovers the nominal limit-cycle (black) an impulse disturbance of mass d_1. The blue trajectory shows the recovery of the nominal limit-cycle, which takes about two periods.

Example – stochastic input

A natural question is whether the feedback-law can also handle a stochastic input, and not just the occasional impulse disturbance. We denote the nominal input rate $r = 17 \cdot 10^3$, and then extend the input to the system to be stochastic, with a uniform distribution from the interval $r(t) \in [0.9, 1.1] \cdot r$.

Should this input be used for the original system, without feedback, the nominal limit-cycle would drift every time the input rate is larger than the nominal one, i.e. when $r(t) > r$. This can be seen in Figure 5.10 where system was run for 1000 periods. The black trajectory shows the nominal limit-cycle, and the blue shows the trajectory of the simulation. One can see that the system does not converge back to the nominal limit-cycle. This implies that the queue-size would grow with t, $q_i(t) \to \infty$, as $t \to \infty$, making it impossible to dimension the necessary buffer size.

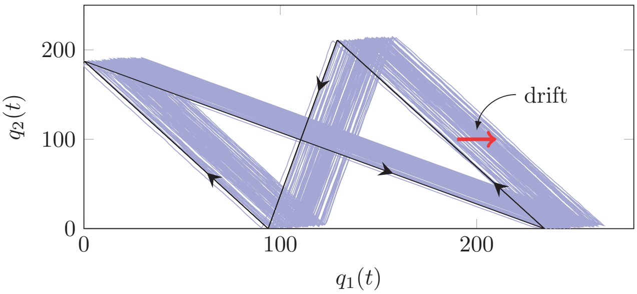


Figure 5.10 State-space trajectory (blue) for a system with a stochastic input but with no feedback from the queue-size. One can see the trajectory drifts away from the nominal limit-cycle (black).

With the feedback-law, however, the system will ensure that the nominal limit-cycle is restored by dynamically changing the on-time for each node every time it is turned on, hence making it possible to dimension the queue-sized. This is illustrated in Figure 5.11 showing the state-space trajectory of the system is centered around the nominal limit-cycle during a simulation length of a 1000 periods.

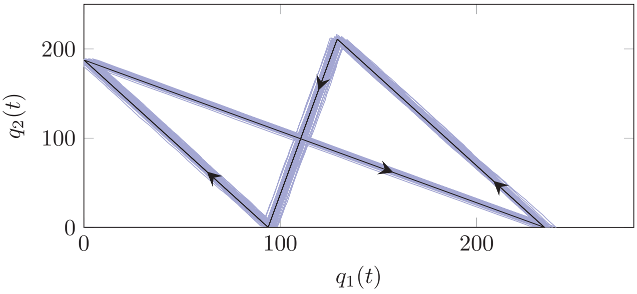


Figure 5.11 State-space trajectory (blue) for a system with a stochastic input where feedback from the queue-size pulls the system back around the nominal limit-cycle.

Monte Carlo Simulation

While the previous subsections were meant to illustrate how to solve the optimization problem as well as to highlight how the feedback would work by bringing

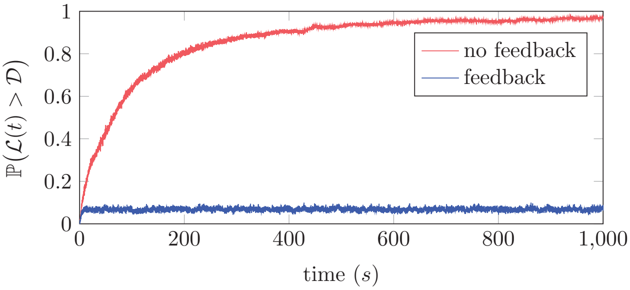


Figure 5.12 A Monte Carlo simulation of 2000 randomly generated service chains, each with 6 nodes. Half of the simulations used feedback (blue) and half do not (red). The y-axis show the probability that one of a packet missing the end-to-end deadline. One can see that the when feedback is used the probability remains below 10% while for the chains without feedback it grows to 100%.

the queue-states back to their nominal trajectories, this subsection focus on a more thorough evaluation using a Monte Carlo simulation.

A total of 2000 randomly generated service chains with 6 nodes were simulated. Half of which used feedback, and the other half did not. Every node was generated randomly with parameters $\mathscr{D} = 0.10$, $\Delta \in [\frac{\mathscr{D}}{10 \cdot m}, \frac{\mathscr{D}}{m}]$, $j^{\mathsf{q}} \in [0.1,\ 1.0]$, $j^{\mathsf{c}} \in [2.0,\ 20.0]$, and $\bar{s}_{\in}[5.0,\ 10.0]$. The mean-value for the input was chosen randomly in the interval $\bar{r} \in [20.0,\ 50.0]$. When running the simulation with stochastic input, during each time-step the input rate to the service chain was uniformly distributed within the interval $r(t) \in [0.8\bar{r},\ 1.2\bar{r}]$ leading to an uncertainty of about 40%.

During the simulations the probability of a chain missing a deadline was computed at each time-step. This probability should be thought of as "*if all the 1000 randomly generated service chains are running simultaneously, what is the probability that one of them will miss a deadline at time t ?*".

The result is shown in Figure 5.12 and illustrates that if one does not use feedback the probability of missing deadlines increases with t and will grow to 100%. The reason is the same as illustrated in Figure 5.10, the queue-states drift and eventually become so large that a large portion of the packets will miss the deadlines. If instead the nodes use feedback to compensate for the uncertainties of the input the probability of a service chain missing a deadline remains below 10%, again highlighting the increased robustness gained when using feedback. The reason why the probability is not reduced to zero with the feedback law, is because of the periodic schedule of the nodes. The nodes can only compensate for an extra amount of traffic in the queue when the additional machine is on. Furthermore, the decision of how long the extra on-time should be, is taken when the additional machine is started. Therefore, should there be more disturbances while the extra machine is on, this will not be accounted for.

5.4 Summary

In this chapter we have developed a general mathematical model for a service-chain residing in a cloud environment. This model includes an input model, a service model, and a cost model. The input-model defines the input-stream of requests to each VNF along with end-to-end deadlines for the requests, meaning that they have to pass through the service-chain before this deadline. In the service-model, we define an abstract model of a VNF, in which requests are processed by a number of machines inside the nodes. It is assumed that each node can change the number of machines that are up and running, but doing so is assumed to take some time. The cost-model defines the cost for allocating compute and storage capacity, and naturally leads to the optimization problem of how to allocate the resources.

The optimization problem for controlling the resources of the nodes in the chain is analyzed and solved under the assumption of a constant input-stream of requests as well as having every node in the chain switch on/off their extra machine with the same period, although not necessarily having the them on for the same duration. The solution derived with these assumptions is then augmented with a feedback-law, allowing the machines to dynamically adjust the necessary on-time for the additional machine depending on whether the queue-size matches the expected queue-size, or whether they deviate. This leads to the system being able to handle both impulse disturbances and stochastic inputs. The difference of using feedback or not is illustrated using both an in-depth example as well as a Monte Carlo simulation, showing that, should the input be stochastic and no feedback is used, the service chain will eventually miss all the deadlines. However, if feedback is used, the probability of missing a deadline remains below 10%.

6

AutoSAC for a chain of nodes

In this chapter we continue to investigate the problem of providing end-to-end deadline guarantees to a chain of nodes. The control strategies presented in this chapter will build on the automatic service and admission controller (AutoSAC) presented for the single node, in Chapter 3, as illustrated in Figure 6.1. However, we will extend it to make better use of the fact that the individual nodes are connected together to form a chain. Comparing Figure 6.1 to the one opening the previous chapter, i.e., Figure 5.1, one can observe this through the re-introduction of the admission controller. Finally, from a high-level perspective, the main difference compared to Chapter 5, is that we now also allow:

a) *dynamic traffic*,

b) *processing uncertainties*,

c) *individual node deadlines*.

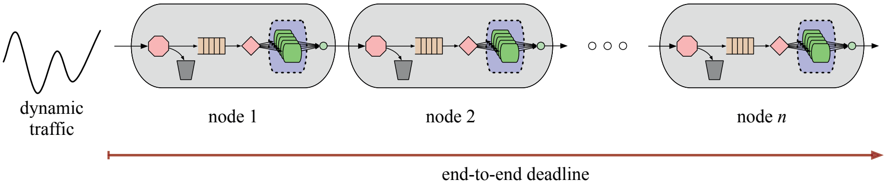


Figure 6.1 Illustration of the chain of cloud services (or virtual network functions) considered in this chapter. The goal is to control how many packets should be admitted into each node, as well as to control how many virtual machines each node should have, in order to ensure that the end-to-end deadline $\mathscr{D}$ of the packets flowing through the chain is met. Note that in this chapter every node also has an admission controller, and that we allow the traffic intensity to vary over time.

6.1 Modeling a chain of nodes

The model used within this chapter is mostly the same as the one presented in Chapter 5. There are, however, a couple of key differences. The first one is that in this chapter we will not consider the cost of allocating computing resources or buffer resources. Instead, we will venture back to use the notion of utility functions presented in Chapter 3. Moreover, we will also introduce a concept for processing uncertainties within the virtual machines.

Concatenation of nodes As illustrated in Figure 6.1 we again have a set of nodes $\mathscr{V}$ with the total number of nodes in the chain given by $n = |\mathscr{V}|$. They are again concatenated, such that for the nodes $i = 2, \dots, n$ the input is equal to the output of the preceding node, $i - 1$:

$$r_i(t) = s_{i-1}(t), \qquad \forall i = 2, 3, \dots, n.$$

Here it should be noted again that we assume there is no communication latency between the nodes. However, it would possible to account for it, and would be necessary should the different nodes reside in different locations, e.g., different data centers. However, adding a communication latency is straightforward, and if such a communication latency (say C) were to be constant between the nodes one could easily account for it by properly decrementing the end-to-end deadline: $\tilde{\mathscr{D}} = \mathscr{D} - C$, and then use the framework developed in this chapter.

Individual node deadline Just as in the Chapter 5, the traffic flowing through the chain must be processed by each of the nodes within an end-to-end deadline $\mathscr{D}$. However, in order to build on what was developed in Chapter 3, we will allow this end-to-end deadline to be split into *individual node deadlines* D_i. This means that traffic passing through the i-th node, should always do so within the individual node deadline.

Naturally, when summing up all of the node deadlines they must remain below the global end-to-end deadline:

$$\sum_{\forall i \in \mathscr{V}} D_i \leq \mathscr{D}. \tag{6.1}$$

One should note here that the this individual node deadline may be different for different nodes in the network, such that $D_i(t) \neq D_{i'}(t)$ if $i \neq i'$. The problem of how to assign these node deadlines will be treated later, in Section 6.2

Machine model Similar to the earlier Chapter 3, the number of virtual machines allocated to a node, $m_i(t) \in \mathbb{Z}^+$, is controlled through a control signal $m_i^{\text{ref}}(t)$. Since we now also allow each node to have an individual time-overhead to start and stop virtual machines, the actual number of active virtual machine in the i-th node is given by

$$m_i(t) = m_i^{\mathsf{ref}}(t - \Delta_i).$$

Just as in the previous chapters, every machine instance running in a node has an *expected service rate* of $\bar{s}$ packets per second. However, the actual performance of the virtual machines may deviate from the expected performance and can depend on where it is deployed as well as on what other processes running on the physical server that the cloud service is hosted on, as shown in [Leitner and Cito, 2016]. To model this, the *maximum processing capacity* of the node at time t is given by:

$$s_i^{\mathrm{cap}}(t) = m_i(t) \cdot \big(\bar{s}_i + \hat{\xi}_i(t)\big), \tag{6.2}$$

where $\hat{\xi}_i(t)$ is the *average machine uncertainty* for the i-th node, given by

$$\hat{\xi}_i(t) = \frac{1}{m_i(t)} \sum_{k=1}^{m_i(t)} \xi_{i,k}(t), \tag{6.3}$$

where $\xi_{i,k}(t)$ is the machine uncertainty for the k-th instance in the i-th node is given by

$$\xi_{i,k}(t) \in [\xi_i^{\mathrm{lb}}, \xi_i^{\mathrm{ub}}] \text{ pps}, \quad -\bar{s}_i < \xi_i^{\mathrm{lb}} \leq \xi_i^{\mathrm{ub}} < \infty,$$

where ξ_i^{lb} and ξ_i^{ub} are the lower and upper bounds of the machine uncertainty, assumed to be known, for instance through benchmarking. This machine uncertainty is assumed to be “fairly constant” during the lifetime of an instance but to change when new instances are started or stopped. By “fairly constant” we mean that it could be modeled as a random process with a small amount of drift during the lifetime of a virtual machine. An example of this is shown in Figure 6.5.

Processing of packets The packets flowing through the chain of nodes are always assumed to be processed in a FIFO manner, and the service rate for the i-th node is again governed by Equation (3.4), with the addition of subscript, giving us:

$$s_i(t) = \begin{cases} s_i^{\mathrm{cap}}(t) & \text{if } q_i(t) > 0, \\ \min\big\{a_i(t),\ s_i^{\mathrm{cap}}(t)\big\} & \text{else.} \end{cases} \tag{6.4}$$

where $a_i(t)$ is admission rate for node i and where $q_i(t)$ is the number of packets in the buffer, given by

$$q_i(t) = A_i(t) - S_i(t), \quad q_i(t) \in \mathbb{R}^+, \tag{6.5}$$

where $A_i(t) = \int_0^t a_i(x)\mathrm{d}x$ is the total amount of packets that has been admitted into the i-th node, and $S_i(t) = \int_0^t s_i(x)\mathrm{d}x$ is the total amount of packets that has been served by the node. Furthermore, it might be useful to also recall that the total amount of packets that has arrived to the i-th node is given by $R_i(t) = \int_0^t r_i(x)\mathrm{d}x$.

Latency and response time. As in the previous chapter, the time that a packet which exits the i-th node at time t has spent inside that node is denoted the *node latency* $L_i(t)$:

$$L_i(t) = \inf\{\tau \geq 0 : A_i(t-\tau) \leq S_i(t)\}, \tag{6.6}$$

with the difference that here we have to consider the difference between $A_i(t-\tau)$ and $S_i(t)$ instead of between $R_i(t-\tau)$ and $S_i(t)$ as we did in the last chapter. The reason for this is that in the previous chapter we did not have any admission control, whereas in this chapter we do.

As in the previous chapters, it is useful to also define the "virtual" latency, or the *expected latency* $\bar{L}_i(t)$. However, due to the machine processing uncertainty of the nodes, this has to be modeled a bit differently compared to the previous chapters. More precisely, in Chapter 3, the expected latency was computed according to

$$\bar{L}_i(t) = \inf\left\{\tau \geq 0 : A_i(t) \leq S_i(t) + \int_t^{t+\tau} m_i(x)\cdot \bar{s}_i \,\mathrm{d}x\right\},$$

but since we in this chapter also consider the processing uncertainty $\hat{\xi}_i(t)$ of the nodes, it must instead be computed according to:

$$\bar{L}_i(t) = \inf\left\{\tau \geq 0 : A_i(t) \leq S_i(t) + \int_t^{t+\tau} m_i(x)\cdot \left(\bar{s}_i + \hat{\xi}_i(x)\right) \mathrm{d}x\right\}. \tag{6.7}$$

One should note that $\bar{L}_i(t)$ is non-causal, and computing it requires information about $m_i(t)$ and $\hat{\xi}_i(t)$ for the future. However, since $m_i(t) = m_i^{\mathrm{ref}}(t-\Delta_i)$, there exists information about $m_i(t)$ up until Δ_i time-units into the future. Moreover, by replacing $\hat{\xi}_i(t)$ with ξ_i^{lb} we can also compute the *worst-case latency* up until Δ_i time-units into the future, which will be done later, in Equation (6.10).

Utility function. In order to capture the performance of the entire chain, we will extend the notion of the utility function presented in Chapter 3. We will therefore define the *average utility* $U(t)$ as the average utility across the nodes in the chain, given by

$$\begin{array}{l} U^{\mathrm{a}}(t) = \frac{1}{n}\sum\limits_{i\in\mathscr{V}} u_i^{\mathrm{a}}(t), \quad U^{\mathrm{e}}(t) = \frac{1}{n}\sum\limits_{i\in\mathscr{V}} u_i^{\mathrm{e}}(t), \\ U(t) = \frac{1}{n}\sum\limits_{i\in\mathscr{V}} u_i^{\mathrm{a}}(t)\cdot u_i^{\mathrm{e}}(t), \end{array} .$$

where the individual utility, availability, and efficiency for the i-th node is given by are given by Equation (3.7), but with the addition of the sub-index:

$$u_i^{\mathrm{a}}(t) = \begin{cases} s_i(t)/r_i(t) & \text{if } L_i(t) \leq D_i \\ 0 & \text{if } L_i(t) > D_i \end{cases}$$

$$u_i^{\mathrm{e}}(t) = \frac{s_i(t)}{s_i^{\mathrm{cap}}(t)}$$

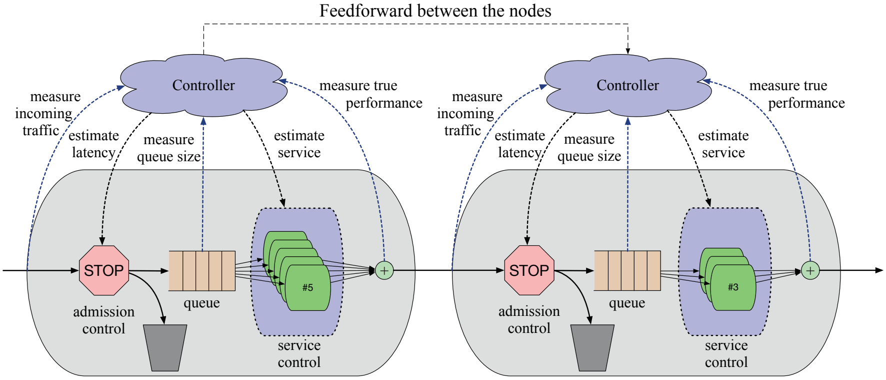


Figure 6.2 Overview of the automatic service and admission-controller highlighting that it uses feedback from the true performance of the nodes to estimate the time it will take incoming packets to pass through the node. It also utilizes feedforward between the nodes to ensure faster reactions to changes in the incoming traffic load.

6.2 Controller design

In this section we extend the AutoSAC presented for a single node in Chapter 3. The main idea is to allow the nodes to also utilize feedforward information between the nodes, as illustrated in Figure 6.2. By also using feedforward and sending information to nodes further down the chain it will allow them to react faster to changes in the incoming traffic. For instance, if the i-th node increases its service rate, it sends a signal to the $(i+1)$-th node letting it know that in Δ_i time-units, it will get an increase in incoming traffic rate. It should be noted that the timing assumptions remain the same as in Chapter 3, and are repeated in Table 6.1.

Property	timing assumption
Long-term trend change of the traffic intensity	1 min – 1 h
Time-overhead Δ_i for changing the number of VMs	1 s – 1 min
End-to-end deadline $\mathscr{D}$ for the packet flow	1 ms – 1 s

Table 6.1 Timing assumptions for the deadline, the change-of-rate of the input, and the overhead for changing the service-rate.

Admission controller

The admission controller in this chapter is very similar to the one presented earlier, in Chapter 3. The difference is that the one presented here, in (6.8), also account for possible machine uncertainty in the nodes. We show in Theorem 6.1 that this admission policy is optimal, and just as in Chapter 3 we also illustrate it with a block diagram, in Figure 6.3. The admission policy is given by:

$$a_i(t) = \begin{cases} r_i(t) & \text{if } A_i(t) < S_i(t) + S_i^{\text{lb}}(t + D_i) - S_i^{\text{lb}}(t) \\ \min\big(r_i(t), s_i^{\text{lb}}(t + D_i)\big) & \text{else,} \end{cases} \tag{6.8}$$

where $S_i^{\text{lb}}(t)$ is given by

$$S_i^{\text{lb}}(t) = \int_0^t m_i(x) \cdot \big(\bar{s}_i + \xi_i^{\text{lb}}\big) \mathrm{d}x. \tag{6.9}$$

The intuition behind the admission policy (6.8) is similar as before; the incoming traffic is guaranteed to meet the node deadline as long as the upper bound on the expected latency is smaller than the node deadline, i.e. as long as $L_i^{\text{ub}}(t) \geq D_i$, with $L_i^{\text{ub}}(t)$ given by:

$$L_i^{\text{ub}}(t) = \inf\left\{ \tau \geq 0 : A_i(t) \leq S_i(t) + \int_t^{t+\tau} m_i(x) \times (\bar{s}_i + \xi_i^{\text{lb}}) \, \mathrm{d}x \right\}. \tag{6.10}$$

It then follows from (6.9) and (6.10), that

$$S_i(t) + S_i^{\text{lb}}(t + D_i) - S_i^{\text{lb}}(t) \geq A_i(t) \quad \Rightarrow \quad L_i^{\text{ub}}(t) \geq D_i.$$

Hence as long as this inequality holds, any incoming packet is guaranteed to meet its deadline, and should thus be admitted, assuming that $r_i(t) < \infty$.

Should there instead be an equality in the expression above, implying that $L_i^{\text{ub}}(t) = D_i$, then care must be taken so that $L_i^{\text{ub}}(t)$ does not grow larger than D_i. Should this happen, as shown in Theorem 6.1, the largest possible admission rate is $a_i(t) \leq s_i^{\text{lb}}(t + D_i)$, since this is the limit of how quickly $S_i^{\text{lb}}(t + D_i)$ grows.

THEOREM 6.1
The admission policy (6.8) will admit as many packets as possible while still ensuring that the admitted packets meet the node deadline. □

Proof It follows from the definition of the upper bound of the delay $L_i^{\text{ub}}(t)$, i.e., by (6.10), that incoming packets will meet their deadlines as long as

$$S_i(t) + S_i^{\text{lb}}(t + D_i) - S_i^{\text{lb}}(t) \geq A_i(t), \tag{6.11}$$

where $S_i^{\text{lb}}(t)$ is given by (6.9). Should there be a strict inequality in (6.11):

$$S_i(t) + S_i^{\text{lb}}(t + D_i) - S_i^{\text{lb}}(t) > A_i(t),$$

then all of the incoming traffic at that time-instance is guaranteed to meet the node deadline D_i. This follows from the fluid approximation which implies that the packet

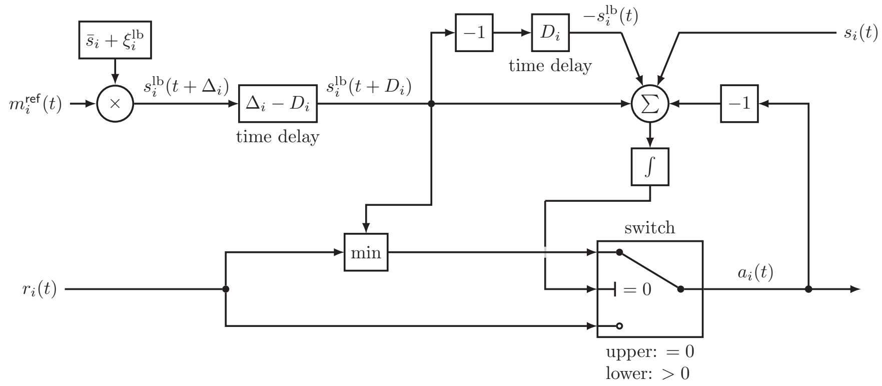


Figure 6.3 Block-diagram of the admission controller when the system has machine uncertainty. The admission control policy can still be computed continuously, although is needs a bit more information. As shown in Theorem 6.1 the feedback-law (or admission policy) derived here is able to admit as many packets as possible, while still guaranteeing that all the admitted packets will meet the node deadline D_i.

size is infinitesimally small. In this case the node should thus admit all the incoming traffic, leading to $a_i(t) = r_i(t)$.

In the case of equality in Equation (6.11), the incoming traffic will only be guaranteed to meet the node deadline as long as the following condition hold:

$$\frac{\partial}{\partial t}\Big\{S_i(t) + S_i^{\mathrm{lb}}(t+D_i) - S_i^{\mathrm{lb}}(t) - A_i(t)\Big\} \geq 0. \tag{6.12}$$

Assuming an admission rate of $a_i(t)$ the above expression can be re-written as

$$\begin{aligned}\frac{\partial}{\partial t}\Big\{S_i(t) + S_i^{\mathrm{lb}}(t+D_i) - S_i^{\mathrm{lb}}(t) - A_i(t)\Big\} = \\ \underbrace{s_i(t)}_{\geq s_i^{\mathrm{lb}}(t)} + s_i^{\mathrm{lb}}(t+D_i) - s_i^{\mathrm{lb}}(t) - \underbrace{a_i(t)}_{\leq s_i^{\mathrm{lb}}(t+D_i)} \geq \\ s_i^{\mathrm{lb}}(t) + s_i^{\mathrm{lb}}(t+D_i) - s_i^{\mathrm{lb}}(t) - s_i^{\mathrm{lb}}(t+D_i) = 0.\end{aligned}$$

From this, one should note that if $a_i(t) > s_i^{\mathrm{lb}}(t+D_i)$ it would lead to a violation of (6.12) since it would lead to it being strictly less than 0. Therefore it is not possible to have a higher $a_i(t)$ and still guarantee that the node deadline of incoming traffic will be met (in the case where $S_i(t) + S_i^{\mathrm{lb}}(t+D_i) - S_i^{\mathrm{lb}}(t) = A_i(t)$).

Finally, assuming the system is initiated with empty queues at time $t = 0$, it follows from the definition of $S_i(t)$, $A_i(t)$, and $S_i^{\mathrm{lb}}(t)$ that

$$\forall t \geq 0,\ \forall i \in \mathscr{V}, \quad S_i(t) + S_i^{\mathrm{lb}}(t+D_i) - S_i^{\mathrm{lb}}(t) \geq A_i(t)$$

since $S_i(0) = A_i(0) = 0$ and they are all non-decreasing. The system will therefore never end up in a state where $L_i^{\mathrm{ub}}(t) > D_i$, thus guaranteeing that

$$\forall t \geq 0,\ \forall i \in \mathscr{V}, \quad L_i^{\mathrm{ub}}(t) \leq D_i. \qquad \square$$

Service controller

In this section we will extend the service controller presented Chapter 3 to also use the feedforward information between the nodes. The intuition is that this will allow the nodes to make a better prediction of the future arrival rate into the node, i.e., to better predict $\hat{r}_i(t)$.

Before that, however, it may be useful to recall that the goal of the service controller is to find the $m_i^{\mathrm{ref}}(t)$ such that the utility function $u_i(t+\Delta_i)$ is maximized. In the previous chapter we showed that the utility function could be approximated with the following expression:

$$u_i(t) \approx \begin{cases} \dfrac{m_i(t) \cdot \left(\bar{s}_i + \hat{\xi}_i(t)\right)}{r_i(t)}, & \text{if } m_i(t) \cdot \left(\bar{s}_i + \hat{\xi}_i(t)\right) \leq r_i(t) \\ \dfrac{r_i(t)}{m_i(t) \cdot \left(\bar{s}_i + \hat{\xi}_i(t)\right)}, & \text{else} \end{cases} \tag{6.13}$$

As mentioned when deriving the expression Equation (3.13), one need information of $\hat{\xi}_i(t+\Delta_i)$ and $r_i(t+\Delta_i)$ in order to maximize $m_i^{\mathrm{ref}}(t)$. However, one can again assume that the machine uncertainty will be fairly constant during Δ_i time-units such that $\hat{\xi}_i(t+\Delta_i) \approx \hat{\xi}_i(t)$.

What thus remains is to derive an estimation of the future arrival rate of the nodes. For the first node in the chain, this will be exactly as in Chapter 3 Equation 6.14:

$$\hat{r}_1(t) = r_1(t) + \Delta_1 \frac{\mathrm{d}r_1(t)}{\mathrm{d}t}.$$

For the subsequent nodes, $i = 2, \ldots, n$ the input-rate might change in a step-wise fashion. This might for instance happen to node $(i+1)$ if the i-th node starts up a new virtual machine instance while having a large amount of traffic in its buffer. In this case, node $(i+1)$ will see a sudden step-wise increase in the rate by which traffic is flowing to it. It is therefore not favorable to use the same estimation for the nodes $i \geq 2$. Instead, we will introduce a feedforward mechanism.

Since $r_i(t) = s_{i-1}(t)$ and $m_{i-1}(x)$ is known for $x \in [0, t+\Delta_{i-1}]$ (with t being the current time) one could estimate the future input-rate $\hat{r}_i(t)$ for nodes $i \geq 2$ with

$$\hat{r}_i(t) \approx \min\left(s_{i-1}^{\mathrm{cap}}(t+\Delta_{i-1}),\ \hat{r}_{i-1}(t)\right), \quad i = 2, \ldots, n.$$

Note that $s_{i-1}^{\mathrm{cap}}(t+\Delta_{i-1})$ is used here, instead of $s_{i-1}^{\mathrm{cap}}(t+\Delta_i)$. The reason is that if $\Delta_i > \Delta_{i-1}$ then one does not have enough information to compute $s_{i-1}^{\mathrm{cap}}(t+\Delta_{i-1})$.

However, one can use the assumption that $\Delta_i \approx \Delta_{i-1}$. Finally, since

$$s_{i-1}^{\mathsf{cap}}(t+\Delta_{i-1}) \approx m_{i-1}^{\mathsf{ref}}(t) \cdot \left(\bar{s}_{i-1} + \hat{\xi}_{i-1}(t)\right)$$

one can summarize the predicted input $\hat{r}_i(t)$ as

$$\hat{r}_i(t) = \begin{cases} r_i(t) + \Delta_i \frac{\mathrm{d}r_i(t)}{\mathrm{d}t}, & i = 1, \\ \min\left\{m_{i-1}^{\mathsf{ref}}(t) \cdot \left(\bar{s}_{i-1} + \hat{\xi}_{i-1}(t)\right),\ \hat{r}_{i-1}(t)\right\}, & \text{else.} \end{cases} \tag{6.14}$$

With Equation (6.14) to predict the incoming traffic of a node, one can again define $\kappa_i(t) \in \mathbb{R}^+$ to be the real-valued number of instances needed to exactly match the predicted incoming rate, just as in Equation 3.15. The only difference is that we now also account for the processing uncertainty $\hat{\xi}_i(t)$:

$$\kappa_i(t) = \frac{\hat{r}_i(t)}{\bar{s}_i + \hat{\xi}_i(t)}. \tag{6.15}$$

The control-law is then again given by Equation 3.16, however slightly adjusted to also account for the node-index:

$$m_i^{\mathsf{ref}}(t) = \begin{cases} \lfloor \kappa_i(t) \rfloor, & \text{if } \lfloor \kappa_i(t) \rfloor \lceil \kappa_i(t) \rceil \geq \kappa_i^2(t), \\ \\ \lceil \kappa_i(t) \rceil, & \text{else.} \end{cases} \tag{6.16}$$

Properties of AutoSAC There are several interesting properties captured by the admission controller and service controller presented in this section. First of all, the admission controller (6.8) ensures, by design, that every packet that is admitted into a node, and thus exits the node, meets its deadline. Therefore, no packets that exit the service-chain will miss their end-to-end deadline.

The service-controller given by equation (6.16) captures both the feedback used from the true performance of the instances (when computing $\hat{\xi}_i(t)$) as well as feed-forward information about future input coming from nodes earlier in the service-chain (when computing $\hat{r}_i(t)$). This makes it robust against machine uncertainties but also ensures that it reacts fast to sudden changes in the input. For instance, given a service-chain of 6 nodes, the 5-th node will know that in Δ_4 time-units, the 4-th node will have $m_4^{\mathsf{ref}}(t)$ instances running and can thus start as many instances as needed to process this new load.

Selection of node deadlines

In this subsection we treat the problem of assigning the node deadlines D_i to the different nodes in the network. Naturally, this assignment is constrained by (6.1):

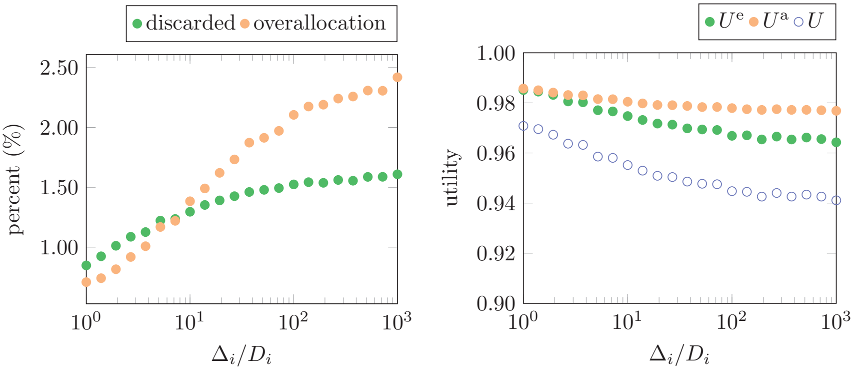
$$\sum_{\forall i \in \mathscr{V}} D_i \leq \mathscr{D}.$$

However, this constraint still leaves one with plenty of options of how to assign the node deadlines. In order to do so in a good way, it might be useful to understand how different choices affect the performance of the system. Naturally, having an unnecessarily short node-deadline may cause it to be unnecessarily difficult to control that node.

The downside of assigning individual node deadlines is that it may introducing some *slack* into the system. By slack we mean that the sum of the local node deadlines for the nodes along the chain might be lower than the end-to-end deadlines.

To gain some understanding in how different node deadlines affect performance of the system we will investigate how different choices affect both the utility metrics as well as how many packets are discarded.

Using the service controller (presented earlier) and the admission controller (presented next) we performed a Monte Carlo simulation with different parameters. The simulation methodology is according to the one presented later in Section 6.3, but where we chose Δ_i for each node instead of randomly generated it, as we will describe later. However, it should be noted the other parameters of the nodes are randomly generated, just as described in Section 6.3. These simulations suggest that there is an interesting relationship between the number of discarded packets and the ratio Δ_i/D_i, as illustrated in Figure 6.4. Note that every data-point in this simulation correspond to the mean value of that metric achieved over 1,000 simulations (using the simulation method presented later in Section 6.3).



(a)

(b)

Figure 6.4 Figure illustrating how the ratio Δ_i/D_i affects the performance of a node. In Figure 6.4(a) one can see how it affects the amount of discarded packets as well as the amount of overallocation. In Figure 6.4(b) one can see how the ratio affects the average availability, efficiency, and utility of the chain. The intuition is that the larger the ratio, the harder it is to have good control of the node since one has to predict the arrival rates further into the future.

The ratio of Δ_i/D_i The intuition gained from Figure 6.4 is that the ratio of Δ_i/D_i is a good metric to optimize for when assigning the node deadlines. This ratio gives insight in how difficult it is to control a node. A small ratio means that it is easy to quickly react to changes of the arrival rate, or to performance changes of the virtual machines. In fact, a ratio smaller than 1 implies that there is always time to react to such changes. As the ratio grows larger, it becomes impossible to react to such changes. Instead, it becomes necessary to have a proactive approach—to predict the future arrival rates. The larger the ratio, the longer into the future one must predict, hence the harder it becomes to control the node.

The optimization problem With the insights about Δ_i/D_i we can set up an optimization problem that assigns node deadlines D_i to every node:

$$\begin{aligned} \text{minimize} \quad & \sum_{\forall i \in \mathscr{V}} \Delta_i/D_i \\ \text{subject to} \quad & \sum_{\forall i \in \mathscr{V}} D_i \leq \mathscr{D} \\ & 0 \leq D_i \qquad \forall i \in \mathscr{V} \end{aligned} \tag{6.17}$$

The optimization problem (6.17) is convex and can thus be solved using standard methods such as Lagrange multipliers or cone programming [Boyd and Vandenberghe, 2004].

We would like to remark here, that other than the time-overhead Δ_i, one could also consider the nominal processing capacity $\bar{s}_i$, or the processing time of packets (if it would be modeled), when assigning the node deadlines. With a larger nominal processing capacity, or a larger processing time of a node might benefit from a larger node deadline. Regarding the larger processing time, the intuition is that with larger nominal processing rate of a virtual machine, the node has more coarse-grained granularity when choosing the number of virtual machines it should have up and running.

6.3 Evaluation

In this section we evaluate how AutoSAC performs when there is a chain of nodes with traffic flowing through it. The simulation method is the same as the one presented in Chapter 3 but with a chain instead of a single node. The main difference compared to Chapter 3 is that we now randomly generated the end-to-end deadline $\mathscr{D}$ as well as the introduction of simulating the machine uncertainty for the nodes. For each simulation the end-to-end deadline was randomly chosen from the interval [250, 500] ms. The nodes in the chain were then assigned their respective node deadline according solution of the to optimal deadline splitting problem (6.17). The machine uncertainty of the nodes was simulated by choosing the bounds randomly

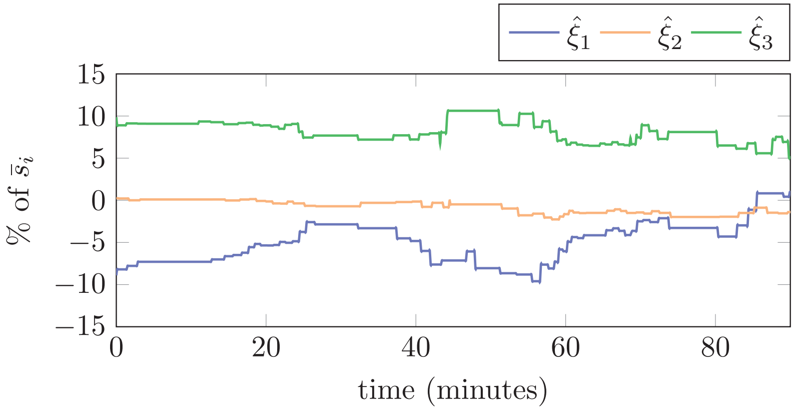


Figure 6.5 The average machine uncertainty, i.e. the difference between the expected and true service performance of the virtual machines, illustrated for each node in the network. It is normalized with respect to the nominal service capacity of one instance, implying that if $\hat{\xi}_i(t) = -10\%$ then the i-th node would need 10% more instances in order to match the incoming traffic load.

from the intervals $\xi_i^{\text{lb}} \in [-0.3 \cdot \bar{s}_i, 0]$ and $\xi_i^{\text{ub}} \in [0, 0.3 \cdot \bar{s}_i]$. During the simulation the machine uncertainty would then be chosen randomly from within these bounds: $\hat{\xi}_i(t) \in [\xi_i^{\text{lb}}, \xi_i^{\text{ub}}]$ every time a new instance was started/stopped in order to simulate the load disturbance generated from not knowing the performance of the physical machine that the instance is launched on. This led to the step-wise behavior shown in Figure 6.5. When the number of instances did not change for a node, the machine uncertainty was simulated as a stochastic process leading to the small drift shown in Figure 6.5.

Next we will show some results for a randomly generated chain with three nodes, and later a comparison with the two “industry”-methods DAS and DOA (as well as their augmented versions with the proposed admission controller).

Example chain

For this example a service chain with three nodes simulated. The end-to-end deadline and the machine uncertainty was simulated as described above, and the rest of the parameters according to Chapter 3. In Figure 6.6 one can see how the different nodes scale the number of instances up/down for the in order to react to the input load. One can also see that they react quickly to each other, which comes from the feedforward-signals between the nodes.

Comparing AutoSAC with state-of-the-art

In this section we compare AutoSAC *dynamic auto-scaling* (DAS) and *dynamic over-provisioning* (DOP), as well as to when they are augmented with the pro-

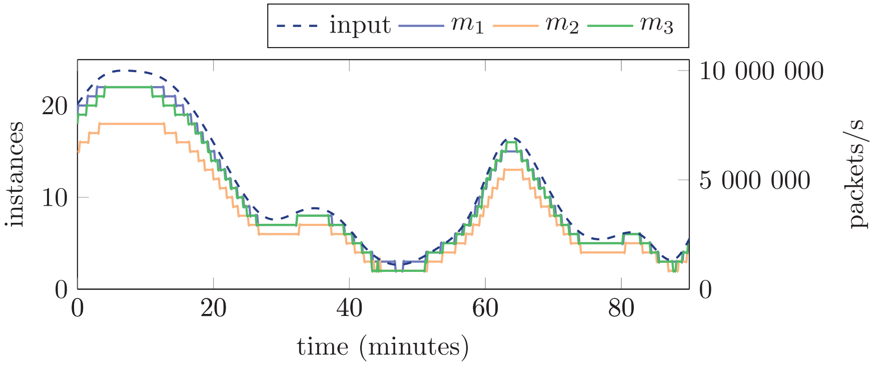


Figure 6.6 Simulation of how the three nodes scale the number of active virtual machines in order to adjust for the changing incoming traffic.

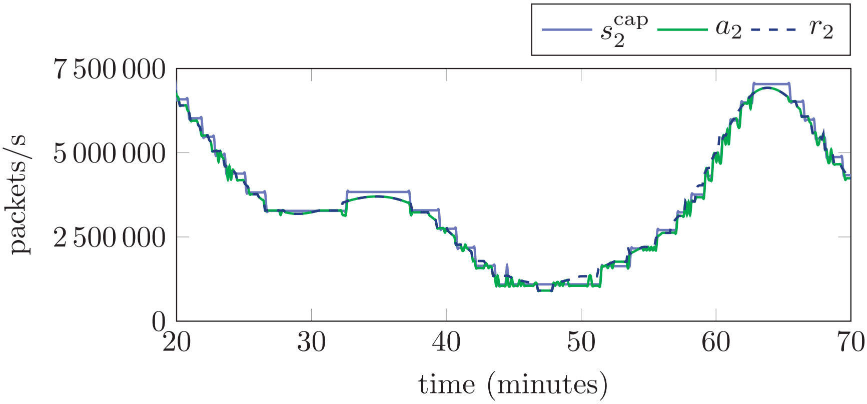


Figure 6.7 Simulation of how the admission rate changes over time for the second node in the chain. It is zoomed in at the interval between 20 min and 70 min in order to illustrate how it handles the varying arrival rate.

posed admission controller (6.8). Just as in Chapter 3 the five methods are compared using a Monte Carlo simulation with 1000 runs each. For every run, we simulated a chain of 4 nodes, where all the parameters were randomly generated (as described earlier). The input data was again randomly selected from the SUNET data. The evaluation of the Monte Carlo simulation is based on the *average utility* $\text{mean}(U) = \frac{1}{t}\int_0^t \sum_{i=1}^n u_i(x)\mathrm{d}x$.

Results The mean of the average utility $U(t)$ for all the simulation runs is presented in Figure 6.8 for each of the five methods. One can see that AutoSAC achieves a utility that is 30–40% better than that of DAS and DOP. The main reason for this is that they are lacking admission control leading to packets missing their deadlines, which eventually results in a low utility.

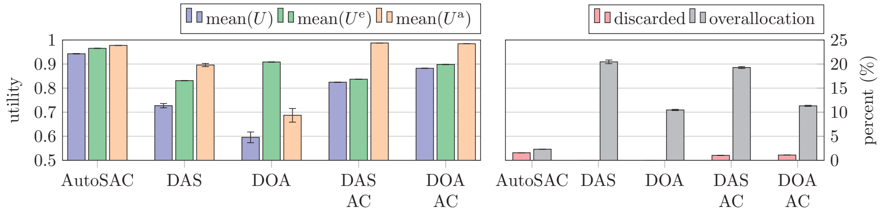


Figure 6.8 Results from the Monte Carlo simulation.

6.4 Summary

In this chapter we have extended mathematical model for a chain of nodes in the cloud. The extended model captures, among other things, the time needed to start/stop virtual resources (e.g., virtual machines or containers), and the uncertainty of the performance of the virtual resources which can deviate from the expected performance due to other tenants running loads on the physical infrastructure. The packets that flow through the forwarding graph must be processed by each of the virtual network functions (VNFs) within some end-to-end deadline.

We have also extended the previously presented AutoSAC (of Chapter 3 to also be cable of handling a chain of nodes. One of the extensions is the added feedforward between the nodes, allowing them to quickly react to changes in the incoming traffic. Furthermore, it also uses feedback from the true performance of the nodes, in order to compensate for any processing uncertainties of the virtual machines in the nodes.

One can note that the AutoSAC presented in Chapter 3 is a special case of the one presented in this chapter, since it is for a single node, and without any machine uncertainty. In the next chapter, 7, it will be further generalized to be able to handle a network of nodes.

7

AutoSAC for a network of nodes

In Chapter 3 we presented AutoSAC for a single-node system. Recall that AutoSAC is a way of combining automatic service and admission control to ensure that traffic flowing through the node met a specific deadline. This approach was then extended to a chain of nodes, in Chapter 6. In this chapter we extend again, in order to allow it to handle a network of nodes with multiple packet flows going through it. A network such as this is illustrated in Figure 7.1, which has five nodes and 4 packet flows traversing the network. Each flow sees a dynamic traffic traversing the network along a specific path. The traffic belonging to a flow must be processed by all nodes along its path within a given end-to-end deadline.

With the introduction of multiple flows, and multiple end-to-end deadlines, the problem of guaranteeing these end-to-end deadlines requires more care than one treating a single flow, as in Chapter 6. The focus of this chapter is therefore on how to account for this when controlling the nodes in the network. Again, the goal is to combine admission control and horizontal scaling of the nodes such that all the flows meet their end-to-end deadlines, all while using as few virtual machines as possible and while discarding as few packets as possible.

The main extensions compared to Chapter 6 are:

a) *Input prediction:* An improvement of the feedforward scheme to allow it to differentiate between “internal” and “external” traffic.

b) *Assignment of node deadlines:* The optimization problem that assigns the node deadlines is generalized to be able to handle multiple flows, and multiple end-to-end deadlines.

c) *Convex utility function:* We present an alternative, convex version, of the utility function.

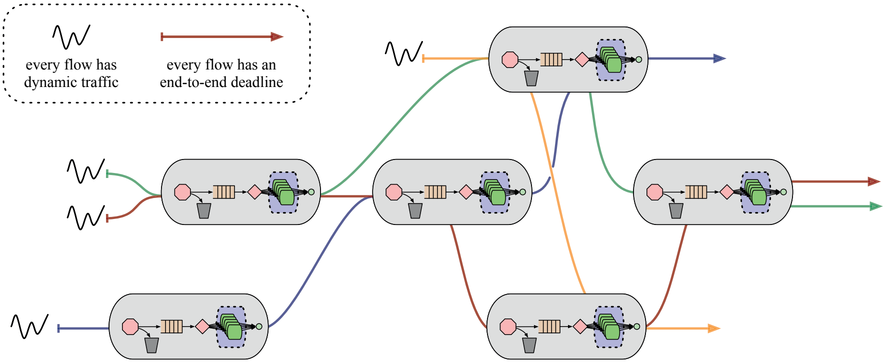

every flow has dynamic traffic
every flow has an end-to-end deadline

Figure 7.1 Illustration of a network of nodes with 4 packet flows passing through it. Each flow has a dynamic traffic as well as an end-to-end deadline.

7.1 Modeling a network of nodes

In this section we extend the mathematical model presented in the previous chapters, and in particular Chapter 6, to allow us to model a network of nodes with multiple packet flows passing through it along distinct paths.

Network and packet flows To model the network of nodes we start by describing the connectivity among them as a time-invariant *directed graph* $\mathscr{G} = \{\mathscr{V}, \mathscr{E}, \mathscr{F}\}$, where

- $\mathscr{V}$ is the set of $n = |\mathscr{V}|$ nodes. For convenience, we label the nodes with the integers from 1 to n, that is $\mathscr{V} = \{1, \ldots, n\}$;
- $\mathscr{E} \subseteq \mathscr{V} \times \mathscr{V}$ is the set of directed edges between these nodes. If $(i, i') \in \mathscr{E}$ then a directed edge from node $i \in \mathscr{V}$ to node $i' \in \mathscr{V}$ exists;
- $\mathscr{F}$ is the set of packet flows traversing the network along specific paths. The path of the j-th flow is modeled by the sequence $p_j : \{1, \ldots, \ell_j\} \to \mathscr{V}$, with $\ell_j \geq 1$ being the length of the path, such that

$$\forall k = 1, \ldots, \ell_j - 1, \quad \big(p_j(k), p_j(k+1)\big) \in \mathscr{E}. \tag{7.1}$$

It should be noted that the function p_j is a mapping from the integers $1, \ldots, \ell_j$ to the set of nodes in $\mathscr{V}$. Hence, $p_j(k)$ is the k-th node of the j-th path. Naturally, Equation (7.1) enforces the existence of an edge between two consecutive nodes in the path of a flow. Since this model is general and allow for flows to traverse a node more than once, it is useful to also define $\delta_{j,i}$ as the number of times that flow j pass through node i. Allowing a flow to traverse the same node multiple times in this way, allow us to capture typical control-applications with feedback-loops.

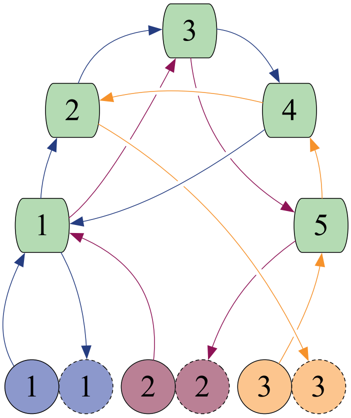


Figure 7.2 A simple network with three sources, three destinations, three cloud functions, and three packet flows $p_1 = \{1,2,3,4,1\}$ (in blue), $p_2 = \{1,3,5\}$ (in red), and $p_3 = \{5,4,2\}$ (in orange).

Example network An example of a network that we can model now is illustrated in Figure 7.2. Using our framework, it can be modeled by a graph $\mathscr{G} = \{\mathscr{V}, \mathscr{E}, \mathscr{F}\}$ with $\mathscr{V} = \{1, 2, 3, 4, 5\}$ and $\mathscr{E} = \{(1, 2), (1, 3), (2, 3), (3, 4), (3, 5), (4, 1), (4, 2), (5, 4)\}$. The paths of three flows are modeled by $p_1 = \{1,2,3,4,1\}$, $p_2 = \{1, 3, 5\}$, and $p_3 = \{5, 4, 2\}$, with end-to-end deadlines given by $\mathscr{D}_1$, $\mathscr{D}_2$, and $\mathscr{D}_3$ respectively. Since the first flow is traversing the first node twice, it means that $\delta_{1,1} = 2$.

Assigning node deadlines Just as in the previous chapter, the goal is to control the admission rate and the service rate of each node in the network such that the number of allocated virtual machines is minimized. To leverage what was previously developed, we will again assign an individual node deadline assigned to each of the nodes in the network. This means that a packet that arrives to the i-th node will only be admitted into the node if it is possible to guarantee that the packet will be processed and exit the function within D_i seconds. This must hold, regardless of which flow the packet belongs to. In other words, every packet arriving to the i-th node will have the same node deadline D_i, even though they might belong to different flows with different *end-to-end deadlines*. Naturally, when assigning these node deadlines, one is constrained by the following:

$$\sum_{\forall i \in p_j} \delta_{j,i} \cdot D_i \leq \mathscr{D}_j, \qquad \forall j \in \mathscr{F}, \tag{7.2}$$

which states that the sum of all the node deadlines over a path j must be less than the end-to-end deadline of that path. Later, in Section 7.2 we will adapt the optimization problem presented in Chapter 6 to ensure it also works for a network of nodes.

7.2 AutoSAC for a network of cloud functions

In this section we extend the previous AutoSAC from Chapters 3 and 6. The main extension can be hinted in the illustration of Figure 7.3 and is a way to distinguish between “external” and “internal” traffic. Along with this we also present a slightly different way to split the end-to-end deadlines of the flows into individual node deadlines. Along with this we will present an alternative control law for a convex utility function.

The intuition behind the new input prediction is that the nodes within the network will share their knowledge of future traffic predictions to their neighbors (which they are sending traffic to). This way, they will use different ways to predict the future traffic depending on whether the traffic is coming from a different node within the network, or if it comes directly from an external source outside the network of nodes. Finally, the timing assumptions for this chapter still remain the same as for in Chapter 6, and is repeated below:

Property	timing assumption
Long-term trend change of the traffic intensity	1 min – 1 h
Time-overhead Δ_i for changing the number of VMs	1 s – 1 min
End-to-end deadlines $\mathscr{D}_j$ for a packet flow	1 ms – 1 s

Table 7.1 Timing assumptions for the deadline, the change-of-rate of the input, and the overhead for changing the service-rate.

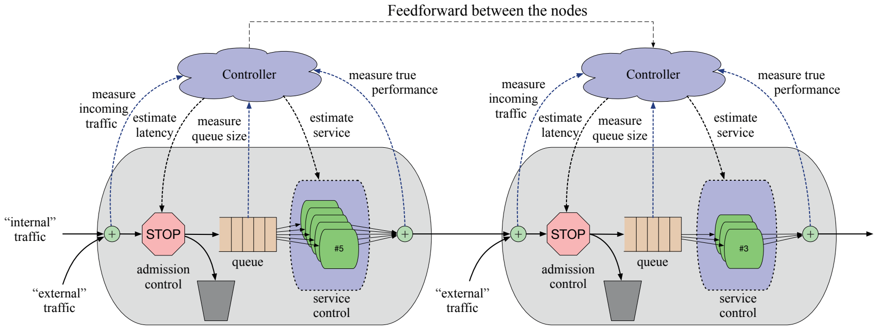


Figure 7.3 Illustration of AutoSAC for a network of nodes. One can in particular see the new addition of distinguishing between external traffic and internal traffic. To predict future internal traffic it uses feedforward information from preceding nodes. To predict external traffic it uses the linear extrapolation used in Chapter 3.

Input prediction

The importance of having a good prediction of arriving traffic comes from the fact that it takes Δ_i seconds for the i-th node to start/stop a virtual machine instance in a node. Recall that the number of instances deployed and running is controlled through $m_i^{\mathsf{ref}}(t)$ and also that $m_i(t) = m_i^{\mathsf{ref}}(t - \Delta_i)$. Hence it takes Δ_i seconds for any changes of $m_i^{\mathsf{ref}}(t)$ to take effect. This means that at time t the node has to make a decision about how many virtual machines will be needed at time $t + \Delta_i$. For this decision to be good, i.e., to yield the highest possible utility, a good prediction of the arrival rate at time $t + \Delta_i$ is necessary.

In order to improve this prediction, we will present a way to differentiate between *external* and *internal* traffic, and use different prediction schemes for each case. The main idea is that some of the traffic that pass through the i-th node might have passed through many other nodes within the network before reaching it. On the contrary, some of the traffic might come directly from an external source outside the network, and arrive at the i-th node without passing through another node in the network. For this reason, introduce the notion of *internal traffic* $r_i^{\mathrm{I}}(t)$ and *external traffic* $r_i^{\mathrm{E}}(t)$. By internal traffic we mean traffic that arrives to the node directly from another node within the network, and by external we mean traffic that arrive directly without passing through another node in the network, i.e, directly from an external source. Together, they form the total arrival rate for the node:

$$r_i(t) = r_i^{\mathrm{I}}(t) + r_i^{\mathrm{E}}(t). \tag{7.3}$$

The distinction between internal and external traffic is useful because it allows us to use different methods when predicting their future arrival rates. We can thus denote the *predicted arrival rate* to the node as $\hat{r}_i(t)$:

$$\hat{r}_i(t) = \hat{r}_i^{\mathrm{I}}(t) + \hat{r}_i^{\mathrm{E}}(t), \tag{7.4}$$

where $\hat{r}_i^{\mathrm{I}}(t)$ and $\hat{r}_i^{\mathrm{E}}(t)$ are the predictions of the internal and the external traffic respectively. Next, the strategies for predicting these two will be described in more detail.

Predicting external traffic When predicting the traffic arriving to the node directly from an external source, the timing assumptions of Table 7.1 imply that it is sufficient to use a linearization-method. The reason is that the rate-of-change of the external traffic is assumed to be on a different time-scale than the time-delay Δ_i. Using linear extrapolation, the prediction of the external traffic can be computed as

$$\hat{r}_i^{\mathrm{E}}(t + \Delta_i) = r_i(t) + \Delta_i \cdot \frac{\mathrm{d}r_i(t + \Delta_i)}{\mathrm{d}t}. \tag{7.5}$$

This is the same estimation used for the node in Chapter 3, or for the first node of the chain in Chapter 6.

Prediction of internal traffic The idea behind the prediction of the internal traffic is to have it as similar to the way traffic was predicted for the nodes $i = 2, \dots, n$ in Chapter 6. However, since we no longer have a single chain of traffic, doing this prediction becomes more involved, and we thus have to consider this for every node-pair that is sending traffic to one another.

Let us begin by denoting the prediction of traffic from node i' to node i as $\hat{r}^{\mathrm{I}}_{(i',i)}(t)$. The prediction of the total internal traffic arriving to node i can then be estimated through

$$\hat{r}^{\mathrm{I}}_{i}(t) = \sum_{i' \in \mathscr{V}} \hat{r}^{\mathrm{I}}_{(i',i)}(t). \tag{7.6}$$

The intuition is that the information about $\hat{r}^{\mathrm{I}}_{(i',i)}(t)$ is sent to node i from node i'. Node i can then simply sum these predictions up to form an estimation about the internal traffic that will arrive in the future.

Computing $\hat{r}^{\mathrm{I}}_{(i',i)}(t)$, which should be done within the i'-th node, can be decomposed into the following two steps:

1. predict the future service-rate $\hat{s}_{i'}(t)$ of node i', and
2. predict the fraction of traffic routed to from node i' to i.

Predicting the future service rate of the i'-th node can be done the same way as was done in Chapter 6 for nodes $i = 2, \dots, n$, i.e., by assuming that it will be the minimum of the maximum service capacity and the incoming traffic of that node:

$$\hat{s}_{i'}(t + \Delta_i) = \min\left(s^{\mathrm{cap}}_{i'}(t + \Delta_i), \hat{r}_{i'}(t + \Delta_i)\right).$$

Along with this the i'-th node also needs to predict what fraction of this future service rate will be routed to the i-th node. By denoting the fraction of traffic which is *currently routed* from node i' to node i by $w_{(i',i)}(t)$, this can be predicted using a similar linear extrapolation as before:

$$\hat{w}_{(i',i)}(t + \Delta_i) = w_{i',i}(t) + \Delta_i \cdot \frac{\mathrm{d}w_{(i',i)}(t)}{\mathrm{d}t}.$$

Finally, by combining the prediction of the future service rate with the fraction of this which will be routed to the i-th node, we arrive at the final expression:

$$\hat{r}^{\mathrm{I}}_{(i',i)}(t + \Delta_i) = \hat{w}_{i',i}(t + \Delta_i) \cdot \hat{s}_{i'}(t + \Delta_i). \tag{7.7}$$

Service control

One of the nice things about the improved predictions presented in Equation (7.6), is that one can still use the service controller presented in Chapter 6 with it. The reason is that this only changes one of the inputs to $\kappa_i(t)$ given in Equation (6.15):

$$\kappa_i(t) = \frac{\hat{r}_i(t)}{\bar{s}_i + \hat{\xi}_i(t)}.$$

The control-law can thus still be computed according to Equation (6.16):

$$m_i^{\mathsf{ref}}(t) = \begin{cases} \lfloor \kappa_i(t) \rfloor, & \text{if } \lfloor \kappa_i(t) \rfloor \lceil \kappa_i(t) \rceil \geq \kappa_i^2(t), \\ \lceil \kappa_i(t) \rceil, & \text{else.} \end{cases}$$

Alternative utility function In the previous chapters the utility function $u_i(t)$ consider the efficiency and the availability to be equally important, $u_i(t) = u_i^{\mathsf{a}}(t) \cdot u_i^{\mathsf{e}}(t)$. Sometimes this might not be desirable, because one might favor efficiency over availability. Therefore, we will here present an alternative utility function, which combine them in a convex way:

$$\tilde{u}_i(t) = \lambda_i \cdot u_i^{\mathsf{a}}(t) + (1 - \lambda_i) \cdot u_i^{\mathsf{e}}(t). \tag{7.8}$$

It might be interesting how this choice of utility function might affect the control-law of (6.16). To do this, we will derive it again, using the same methodology as in Chapter 3 and 6. By using the approximation methodology as before, one finds that the number of machine instances that maximizes the alternative utility function is given by the following:

$$m_i^{\mathsf{ref}}(t) = \begin{cases} \underset{x \in \mathbb{Z}^+}{\arg\max} \left\{ \lambda_i \dfrac{x}{\kappa_i(t)} + (1 - \lambda_i) \right\}, & \text{if } x \leq \kappa_i(t), \\ \underset{x \in \mathbb{Z}^+}{\arg\max} \left\{ \lambda_i + (1 - \lambda_i) \cdot \dfrac{\kappa_i(t)}{x} \right\}, & \text{else.} \end{cases} \tag{7.9}$$

This follows since

$$\tilde{u}_i(t) = \lambda_i \cdot u_i^{\mathsf{a}}(t) + (1 - \lambda_i) \cdot u_i^{\mathsf{e}}(t) = \lambda_i \cdot \frac{s_i(t)}{r_i(t)} + (1 - \lambda_i) \cdot \frac{s_i(t)}{s_i^{\text{cap}}(t)}.$$

By then approximating $s_i(t)$ by $s_i(t) \approx \min\left(s_i^{\text{cap}}(t), r_i(t)\right)$ and $r_i(t) \approx \hat{r}_i(t)$ one get

$$\tilde{u}_i(t) \approx \begin{cases} \lambda_i \cdot \dfrac{s_i^{\text{cap}}(t)}{\hat{r}_i(t)} + (1 - \lambda_i) \cdot \dfrac{s_i^{\text{cap}}(t)}{s_i^{\text{cap}}(t)} & \text{if } s_i^{\text{cap}}(t) \leq r_i(t) \\ \lambda_i \cdot \dfrac{\hat{r}_i(t)}{\hat{r}_i(t)} + (1 - \lambda_i) \cdot \dfrac{\hat{r}_i(t)}{s_i^{\text{cap}}(t)} & \text{else .} \end{cases}$$

And finally, since

$$\frac{\hat{r}_i(t)}{s_i^{\text{cap}}(t)} = \frac{\hat{r}_i(t)}{m_i(t) \cdot \left(\bar{s}_i + \hat{\xi}_i(t)\right)} = \frac{\kappa_i(t)}{m_i(t)},$$

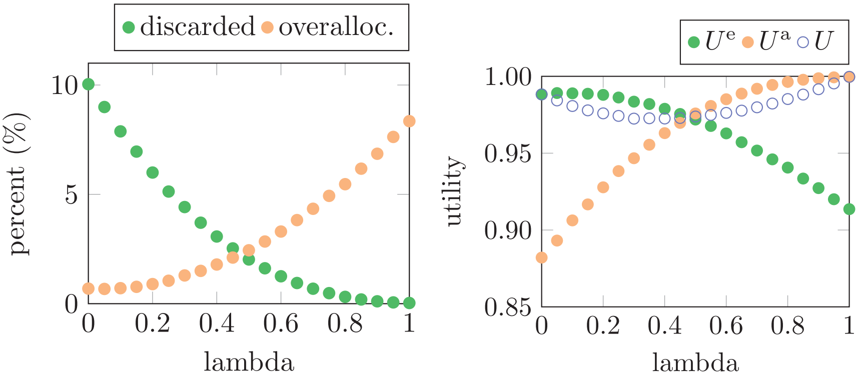
we finally arrive at Equation (7.9). One can see that the upper case of Equation (7.9) is maximized when x is as large as possible within that case, i.e. with $x = \lfloor \kappa_i(t) \rfloor$, while the lower case is maximized when x is as small as possible, i.e. with $x =$

$\lceil \kappa_i(t) \rceil$. The remaining question is therefore which of the two cases that yield the largest utility. One can verify that this indeed evaluates to the following expression:

$$m_i^{\mathrm{ref}}(t) = \begin{cases} \lfloor \kappa_i(t) \rfloor, & \text{if } \lambda_i \lfloor \kappa_i(t) \rfloor \lceil \kappa_i(t) \rceil + (1-2\lambda_i)\kappa_i(t) \lceil \kappa_i(t) \rceil \geq (1-\lambda_i)\kappa_i^2(t), \\ \lceil \kappa_i(t) \rceil, & \text{else.} \end{cases} \tag{7.10}$$

where again $\kappa_i(t) = \hat{r}_i(t)/\left(\bar{s}_i + \hat{\xi}_i(t)\right)$. Comparing the two control-laws (3.16) and (7.10) one can see that the alternative control-law (7.10) is equivalent to (3.16) when the efficiency and availability are considered equally important, i.e. when $\lambda_i = 1/2$.

To give some intuition about how different choices of λ affect the system, we have run some simulations, according to the simulation methodology described later, in Section 7.3. The result of these is shown below, in Figure 7.4. One can clearly see that a low lambda favors a high efficiency, but leads to more packets being discarded. Vice verse, a high lambda favors a high availability, but leads to larger overallocation of processing capacity. Interestingly enough, the most difficult choice of lambda, with respect to attaining a high utility, is when $\lambda = 0.5$.



(a)

(b)

Figure 7.4 Figure illustrating how the choice of lambda affect the performance when the alternative utility function is used to derive the alternative control-law, presented in (7.10).

Assignment of node deadlines

In order to adapt AutoSAC to handle a network of nodes, we must also adapt how the node deadlines are assigned. Therefore, the optimization problem (6.17) for the node deadline assignment, presented in Chapter 6, will be generalized here. The main difference is that the set of feasible node deadlines is now constrained by multiple end-to-end deadlines, instead of just the one.

Having more end-to-end deadlines to consider, will likely lead to more slack being introduced in the system. Recall that by slack, we meant the sum of the node deadlines for the nodes along a path might be lower than the end-to-end deadlines of that path:

$$\text{slack} = \mathscr{D}_j - \sum_{\forall i \in p_j} \delta_{j,i} \cdot D_i.$$

The optimization problem When considering the constraints added by the multiple end-to-end deadlines, the optimization problem (6.17), then becomes:

$$\begin{aligned}
&\text{minimize} && \sum_{\forall i \in \mathscr{V}} \Delta_i / D_i && \\
&\text{subject to} && \sum_{\forall i \in p_j} \delta_{j,i} D_i \leq \mathscr{D}_j && \forall j \in \mathscr{F} \\
& && D_i \geq 0 && \forall i \in \mathscr{V}
\end{aligned} \tag{7.11}$$

where $\delta_{j,i}$ indicates how many times path j goes through node i and Δ_i indicates the time-overhead necessary to change the number of instances running in the i-th node. One should note that the optimization problem (7.11) is still convex and can thus still be solved using standard methods such as Lagrange multipliers [Boyd and Vandenberghe, 2004].

Solving it with Lagrange multipliers As mentioned in earlier one can solve the optimization problem (7.11) using Lagrange multipliers. Disregarding the final constraint of (7.11), which is possible due to the convex nature of the problem, one arrives at the following Lagrangian:

$$\Omega = \sum_{\forall i \in \mathscr{V}} \frac{\Delta_i}{D_i} + \sum_{j=1}^{f} \mu_j \cdot \Big(\sum_{\forall i \in p_j} \delta_{j,i} D_i - \mathscr{D}_j \Big) \tag{7.12}$$

where D_i and μ_j are the primal and dual optimization variables. The Karush-Kuhn-Tucker conditions for (7.12) are:

$$\begin{aligned}
&\frac{\partial \Omega}{\partial D_i} = -\frac{\Delta_i}{D_i^2} + \sum_{j=1}^{f} \mu_j \delta_{j,i} = 0 && \forall i \in \mathscr{V} \\
&\mu_j \cdot \Big(\sum_{\forall i \in p_j} \delta_{j,i} D_i - \mathscr{D}_j \Big) = 0 && \forall j \in \mathscr{F} \\
&\mu_j \geq 0 && \forall j \in \mathscr{F}
\end{aligned} \tag{7.13}$$

It should be noted that when solving (7.13) one will likely end up with many possible cases for μ_j and D_i, however, only a single set of parameters will yield an optimal value for (7.12), which can be denoted as (D_i^*, μ_j^*).

Perturbation analysis Out of curiosity it might be interesting to investigate what interval of parameters that will still preserve the solution in the same set. Or in other words, *how can one change D_i or $\mathscr{D}_j$ without changing the set of paths where the end-to-end deadline constraints are active?*

To do this we start by denoting the set of paths where the end-to-end deadlines constraints are active as the set of J:

$$J \subseteq \mathscr{F} \quad : \quad \forall j : \sum_{\forall i \in p_j} \delta_{j,i} D_i = \mathscr{D}_j \tag{7.14}$$

With this set, one will end up with the following set of equations:

$$\begin{aligned}
& -\frac{\Delta_i}{D_i^2} + \sum_{\forall j \in J} \mu_j \delta_{j,i} = 0 && \forall i \in \mathscr{V} \quad (n \text{ eqs.}) \\
& \sum_{\forall i \in p_j} \delta_{j,i} D_i = \mathscr{D}_j && \forall j \in J \quad (|J| \text{ eqs.}) \\
& \mu_j = 0 && \forall j \notin J \quad (|\mathscr{F}| - |J| \text{ eqs.})
\end{aligned} \tag{7.15}$$

Doing some small perturbation analysis around this we wish to investigate what intervals of parameters still preserve the solution within the same set of J. We thus assume that the perturbations will not change which μ_j's are active. The perturbations we introduce into the equations of (7.15) are ΔD_i, $\Delta \mathscr{D}_j$, and $\Delta \mu_j$. Here it should be noted that the perturbation Δ should not be confused with the time-overhead to start a new instance, i.e., with Δ_i. When referring to the time-overhead within this chapter we will always use a subscript index, and when referring to the perturbation we will not use a subscript index. With these perturbations we arrive at the following set of equations:

$$\begin{aligned}
& -\frac{\Delta_i}{(D_i + \Delta D_i)^2} + \sum_{\forall j \in J} \delta_{j,i} \cdot (\mu_j + \Delta \mu_j) = 0 && \forall i \in \mathscr{V} \\
& \sum_{\forall i \in p_j} \delta_{j,i} (D_i + \Delta D_i) = \mathscr{D}_j + \Delta \mathscr{D}_j && \forall j \in J \\
& \mu_j + \Delta \mu_j = 0 && \forall j \notin J
\end{aligned} \tag{7.16}$$

Assuming small ΔD_i we can then neglect higher order terms of ΔD_i and thus approximate the first equation of (7.16) according to

$$-\frac{\Delta_i}{(D_i + \Delta D_i)^2} \approx -\frac{\Delta_i}{D_i^2 + 2 D_i \Delta_i} \approx \frac{\Delta_i}{D_i^2 \cdot (1 + 2\frac{1}{D_i} \Delta D_i)} \approx -\frac{\Delta_i}{D_i^2} \cdot \left(1 - 2\frac{1}{D_i} \Delta D_i\right)$$

If we then continue by inserting this into the equations of (7.16) we arrive at the following set of equations

$$\begin{aligned} &\cancel{-\frac{\Delta_i}{D_i^2}} + 2\frac{\Delta_i}{D_i^3}\Delta D_i + \cancel{\sum_{\forall j\in J}\mu_j\delta_{j,i}} + \sum_{\forall j\in J}\delta_{j,i}\Delta\mu_j = 0 && \forall i\in\mathscr{V}\\ &\cancel{\sum_{\forall i\in\mathscr{V}}\delta_{j,i}D_i} + \sum_{\forall i\in\mathscr{V}}\delta_{j,i}\Delta D_i = \cancel{\mathscr{D}_j} + \Delta\mathscr{D}_j && \forall j\in J \end{aligned}$$

where some of the terms cancel out due to the solutions of (7.15) along with the assumptions that the perturbations are small enough to not change the set J of the solution. This finally leads to the following ***linear relationship*** for the perturbations:

$$\left[\begin{array}{c|c} \mathbb{D} & P_J^T \\ \hline P_J & \mathbf{0} \end{array}\right] \cdot \left[\begin{array}{c} \Delta\mathbf{D} \\ \hline \Delta\mu \end{array}\right] = \left[\begin{array}{c} \mathbf{0} \\ \hline \Delta\mathscr{D} \end{array}\right] \tag{7.17}$$

where $\mathbb{D}$ is a diagonal matrix $\mathbb{D} = 2\cdot\mathrm{diag}(\Delta_i/D_i^3) \in \mathbb{R}^{n\times n}$, $\Delta\mathbf{D}$ correspond to the vector $\Delta\mathbf{D} = [\Delta D_1, \Delta D_2, \ldots, \Delta D_n]^T$, and finally where $P_J \in \mathbb{N}^{|J|\times n}$ is the incidence matrix where the rows correspond to the flows $j\in J$, and the columns to the nodes $i\in\mathscr{V}$, such that $P_{j,i} = \delta_{j,i}$.

The equation (7.17) gives us insight about how changing μ or $\mathbf{D}$ causes changes in $\mathscr{D}$, or vice verse. One should note that the relationship between $\Delta\mathbf{D}$ and $\mathscr{D}$ is linear, with the constant given by P_J. Should one instead be interested in how changes of $\mathscr{D}$ affect μ one has to solve for it in the equation system above. Due to the blockwise structure of (7.17) one will arrive at:

$$\mathbb{D}\Delta\mathbf{D} + P_J^T\Delta\mu = \mathbf{0} \quad \Leftrightarrow \quad \Delta\mathbf{D} = -\mathbb{D}^{-1}P_J^T\Delta\mu.$$

The other block-equality in (7.17) can then be expanded as follows:

$$\begin{aligned} &P_J\Delta\mathbf{D} = \Delta\mathscr{D}\\ \Rightarrow &-P_J\mathbb{D}^{-1}P_J^T\Delta\mu = \Delta\mathscr{D}\\ \Rightarrow &\Delta\mu = -\left(P_J\mathbb{D}^{-1}P_J^T\right)^{-1}\cdot\Delta\mathscr{D} \end{aligned}$$

where the final step assumes that $(P_J P_J^T)$ is invertible.

7.3 Evaluation

To evaluate the performance of the generalized AutoSAC we perform a similar Monte Carlo simulation as in Chapters 3 and 6. The main difference here is that we now simulate a network of nodes, but still where the parameters are randomly generated. The network used as a basis for the evaluation is the one depicted in Figure 7.2, i.e., with five nodes and three paths.

For each simulation the input rate for each flow in the network was generated just as in Chapters 3 and 6, i.e., by randomly selecting a 100 minute interval of the SUNET-data, and then scaling it such that it had a peak of 10 000 000 packets per second. The nominal service rates $\bar{s}_i$ of the nodes in the network were to be randomly chosen from the interval of $[1\,000\,000,\ 2\,000\,000]$ packets per second. The reason for the larger nominal service-rate is to allow the nodes to have roughly the same number of virtual machines as in the simulations in Chapters 3 and 6. The end-to-end deadlines $\mathscr{D}_j$ for the flows in the network was randomly chosen from the interval of $[250,\ 500]$ ms. The individual node deadlines were then assigned according to the solution of the optimization problem (7.11).

Example simulation In Figures 7.5–7.6 we show some results from one of the many simulations performed during the Monte Carlo simulations. In Figure 7.5 one can see how the arrival rate for the three paths in network varies over time, and in Figure 7.6 one can see how the nodes adjust their maximum processing capacity as well as how the admission rate adjusts. This Figure is zoomed in during the interval between $t = 20$ min and $t = 60$ min, in order to illustrate this a bit clearer.

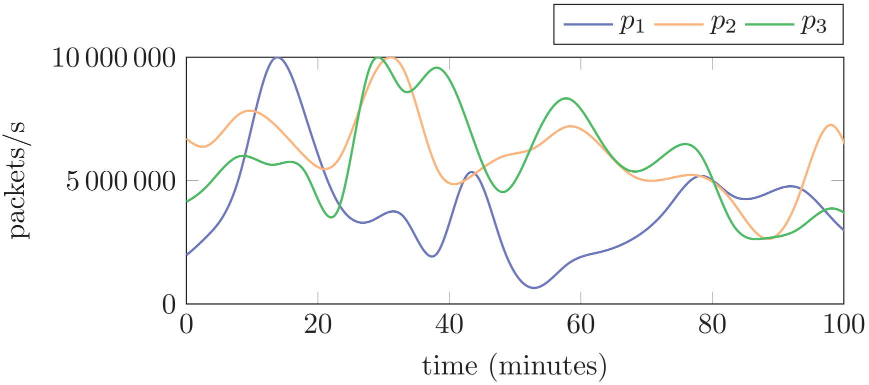


Figure 7.5 The arrival rate for the three paths during one of the simulations.

Comparison with state-of-the-art Just as in the previous chapters, AutoSAC is compared against the industry methods DAS and DOA, along with the versions of when they are augmented with the admission controller of AutoSAC. Each method was again simulated 1000 times. The network used during these simulations are again the one illustrated in Figure 7.2.

Results The result of the Monte Carlo simulation is shown in Figure 7.7. The left part shows the comparison of the average utility, efficiency, and availability. The right part illustrates the same result, but by instead showing the fraction of packets that are discarded as well as how much overallocation each method cause. One

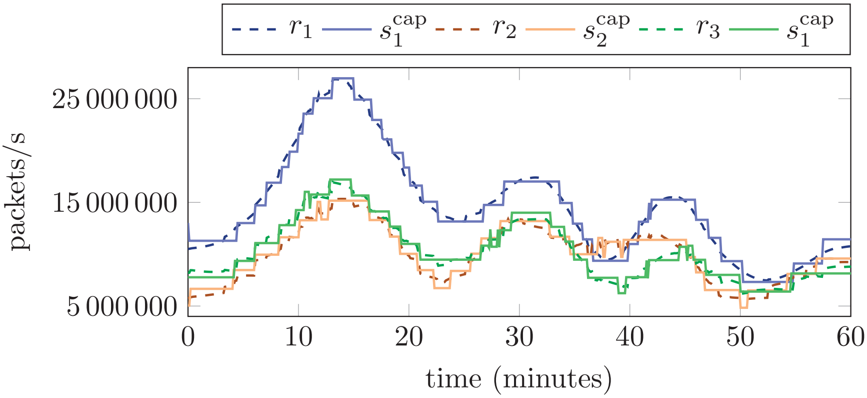


Figure 7.6 Simulation of how the admission rate changes over time for the three nodes in the network. It is zoomed in at the interval between $t = 20$ min and $t = 60$ min in order to illustrate how it handles the varying arrival rate.

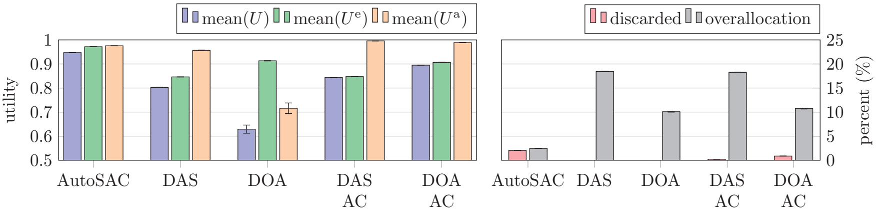


Figure 7.7 Through a large number of simulations of the network depicted in Figure 7.2 the performance of AutoSAC developed in this chapter was compared with what is currently used in industry. The methods AutoSAC was compared against were dynamic auto-scaling (DAS) and dynamic overallocation (DOA). Neither DAS nor DOA has admission control so they were augmented with the one developed in this chapter. In the left figure one can see the result of the average utility, efficiency, and availability for each of the five methods, and in the right figure the fraction of the incoming packets that are discarded and the average amount of overallocation of processing capacity.

can see that AutoSAC is again close to optimal in the average utility, availability, and efficiency. As expected, DAS has an efficiency in the range of 0.8–0.9, since those are the thresholds by which it bases its auto-scaling on. It should be noted that increasing this band of thresholds did not improve the efficiency (the current range yielded the best performance). DOA achieves an efficiency of around 0.75. The reason for the poor average utility for these two methods, however, is the lack of an admission controller. Without one, a queue will build up every time there is a lack of processing capacity. This in turn increases the latency, causing packets to miss their deadlines resulting in a low availability. By augmenting DAS and DOA with the admission controller of AutoSAC, the availability is significantly increased by allowing it to drop some of the packets in a strategic way. By looking at the right part of Figure 7.7 one realizes that it is only a very small fraction of the packets that are actually discarded, so it is a sacrifice well worth making.

7.4 Summary

In this chapter we have derived a general mathematical model for networks of nodes. Looking back at AutoSAC in Chapter 3 and 6 one can realize that they are both special cases of the AutoSAC proposed within this chapter. The main generalization is that now every node in the network can use both feedforward information from other nodes in the network, as well as to predict future external network.

Along with the generalization of AutoSAC in this chapter, was also a generalization of the deadline assignment problem. This was also analyzed to see how changes to the different end-to-end deadlines of the flows passing through the network would affect the assignments.

Finally, we presented a generalization of the utility function, allowing the user to favor either efficiency or availability by using a convex combination of the two. We derived a control-law for this alternative utility function, and showed that when efficiency and availability are considered equally important, then this new control-law is exactly the same as the previous control-law.

In the next chapter we will show how AutoSAC provides the foundation to allow a network to dynamically change the assigned node deadlines over time, which in turn allow new flows to join the network, as well as existing nodes to leave the network. In essence, this allows for a dynamic network topology.

8

End-to-end deadlines over dynamic network topologies

In the chapters leading up to this we have treated the problem of how to control individual nodes in order to give global guarantees. We have hinted slightly about taking a step back and perform global control when we introduced the optimization problem of how to assign individual node deadlines in Chapters 6 and 7. In this chapter we will take the final step on the journey of this thesis, and propose a way which allows us to guarantee that the end-to-end deadlines of flows passing through the network will always be met, even when the network topology is changing over time, as illustrated in Figure 8.1.

This approach of this chapter differs a bit from the previous chapters, because now the focus is on how to control the node deadlines over time, as well as how the nodes and the flows of the network are allowed to join and leave. To ensure that the end-to-end deadlines are met, we also propose a set of protocols for how to dynamically control the node deadlines, as well as to control how nodes and flows are allowed to join and leave the network over time. We formally prove that these protocols work as intended in Theorem 8.1 and Corollary 8.1.

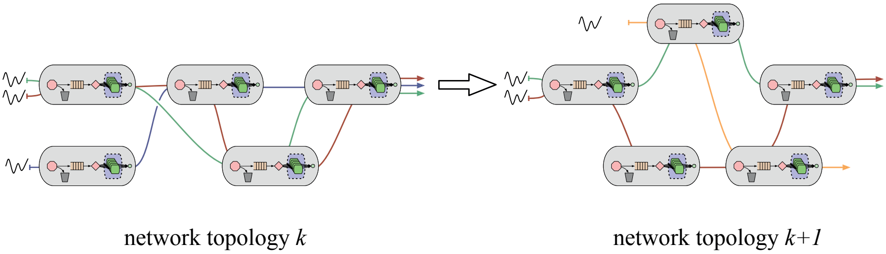


Figure 8.1 Illustration of how the network topology is allowed to change over time. In this chapter we propose a way to allow for this, yet still guarantee that the end-to-end deadlines of the flows traversing the network are met.

8.1 The system model

The focus of this chapter is to develop a framework where applications are implemented as flows of packets through a network of nodes. Each node offers a service to the incoming packets. The goal is then to manage the on-line arrival/departure of flows and the on-line arrival/departure of nodes of the network in a way such that end-to-end deadlines of flows are honored, while still allowing for topology changes.

Section 8.1 presents definitions and assumptions with respect to the services offered by *nodes*. Applications are then defined as a set of interconnected services and referred to as *flows*. Definitions and assumptions on flows are given in Section 8.1. Finally, in Section 8.1, we present some assumptions on how these flows and nodes interact with the resource manager.

Model of the nodes

A node represents an entity that offers a service to the incoming packets. The time taken by a node to provide the service to an incoming packet is given in Definition 8.1 below. The nodes in the system are denoted by $\mathscr{V}(t) \subset \mathbb{N}$, which is the set of indexes of the nodes present at time t. The terms "node" and "vertex" are used interchangeably[1].

DEFINITION 8.1—RESPONSE TIME

We define the *response time* $L_i(t)$, as the time taken by a packet entering node i at time t to be processed. □

It should be noted that the new definition of $L_i(t)$ is a generalization of the definition given in Chapters 3–7, since from Definition 8.1 we can remark the following facts:

- The response time $L_i(t)$ accounts for all sources of delay possibly occurring within node i (queuing, interference, processing, etc.).
- All incoming packets are treated in the same way regardless of the flow they belong to. Differentiating packets depending on the application they belong to is feasible. This would require us to attach the flow index j to the response time, $L_{i,j}(t)$. However, this choice poses only a notational challenge with no conceptual added value. For this reason we believe that letting the packet response-time depend only on the node (and time) does not significantly impact the applicability of the results. We leave the case of per-flow response time to future works.
- Moreover, in contrast to a large body of the research on real-time systems, this chapter does not address how the response time may be computed based

[1] We may use the term "node" when we refer to its capacity to process packets, while the term "vertex" is more often used when referring to the topological structure of the network.

on the amount of incoming workload (due to the arrival of packets) and the amount of processing capacity of a node, etc. The interested reader can refer to a vast relate literature addressing this aspect [Klein et al., 2014; Nylander et al., 2018; Henriksson et al., 2004; Millnert et al., 2017; Millnert et al., 2018; Padala et al., 2007; Voigt and Gunningberg, 2002; Gandhi et al., 2002].

Instead, we assume that the information of $L_i(t)$ is available, and the focus is instead on the interactions between nodes aimed at guaranteeing end-to-end deadlines in the context of a dynamic network.

Naturally, the service provided to packets by a node might include some minimum requirements, which determines a lower bound $\underline{L}_i$ to the response time of the service time of a node. The node is unable to process packets in less time than this value. Hence, to model the minimum time needed by a node to process a packet, we introduce the following definition.

DEFINITION 8.2—RESPONSE TIME LOWER BOUND
We define the *response time lower bound* $\underline{L}_i$, as the minimum a packet may take to be processed by node i.

In practice, $\underline{L}_i$ may represent the pure processing time of a packet, without any interference or delay of any kind.

Finally, we assume that the node is capable of *controlling its response-time*, as stated below in Assumption 8.1. This assumption is exactly what was treated in Chapters 3–7, but is also backed by the vast body of research for different ways of controlling the response-time of cloud services. For instance, in [Nylander et al., 2018] they use the concept of "brownout control" to ensure that the response-time of a server is within a desired limit. In [Henriksson et al., 2004] control theory is used to modify the processing capacity of the web servers to control response times. More interesting work addressing this is found in [Padala et al., 2007; Voigt and Gunningberg, 2002; Gandhi et al., 2002].

ASSUMPTION 8.1 (RESPONSE-TIME CONTROL) We assume that a node i is capable of controlling its response-time $L_i(t)$ such that it is always below a deadline $D_i(t)$:

$$\forall t \geq 0,\ \forall i \in \mathscr{V}(t), \quad \underline{L}_i \leq L_i(t) \leq D_i(t). \tag{8.1}$$

□

It should be noted that Assumption 8.1 gives the implicit requirement that the node deadlines always have to be larger than the lower-bound of the response-time:

$$\forall t \geq 0, \forall i \in \mathscr{V}, \quad \underline{L}_i \leq D_i(t). \tag{8.2}$$

Model of the flows

An application in the system is modeled as a *flow* indexed by some $j \in \mathscr{F}(t)$ and characterized by a *path* and an *end-to-end deadline*, properly defined next.

DEFINITION 8.3—PATH
The *path* of a flow $j \in \mathscr{F}(t)$ is defined by the sequence $p_j : \{1,\dots,\ell_j\} \to \mathscr{V}(t)$, with $\ell_j \geq 1$ being the *length of the path*, such that

$$\forall i = 1,\dots,\ell_j - 1, \quad (p_j(i), p_j(i+1)) \in \mathscr{E}(t), \tag{8.3}$$

where $\mathscr{E}(t) \in \mathscr{V}(t) \times \mathscr{V}(t)$ are the current edges between the nodes of the network.□

Note that this definition is similar to the one presented in Chapter 7, but also allows for the set of flows in the network to change over time.

By Definition 8.3, it follows that $p_j(i)$ is the i-th node on the path of flow j. It should be noted that Equation (8.3) enforces the existence of an edge of the graph between two consecutive nodes in a path. It should be noted that this path may traverse a node more than once, which allows us to capture typical client-server sessions. With a slight abuse of notation, we may denote the image of the map p_j, which is the set of nodes touched by path j, by p_j only, rather than $p_j(\{1,\dots,\ell_j\})$.

We remark that given the type of applications considered in this thesis (e.g., cloud microservices or virtual network functions), each node is considered to provide a unique service to the packets. Hence, the route of packets belonging to a flow is assumed to be known and does not have to be decided at run time.

As in the Chapters 5–7, every flow is assumed to have an end-to-end deadline, however, with Definition 8.4 we again allow the set of flows to change dynamically over time.

DEFINITION 8.4—END-TO-END DEADLINE
We define the *end-to-end deadline* $\mathscr{D}_j$ of the flow $j \in \mathscr{F}(t)$ as the maximum time a packet may take to be processed by all nodes along the path p_j. □

Finally, a flow j is also characterized by an *end-to-end response-time* $\mathscr{L}_j(t)$, which is the time taken by a packet entering the first node $p_j(1)$ of the j-th flow at time t to be processed by all nodes along its path p_j. However, before properly defining $\mathscr{L}_j(t)$, we need to introduce the *mid-path response-time* $\mathscr{L}_{j,i}(t)$, that is the time needed by a packet that entered flow j at time t to pass through the first i nodes of the path p_j. Formally, this quantity is defined recursively by

$$\mathscr{L}_{j,i}(t) = \begin{cases} L_{p_j(1)}(t) & \text{if } i = 1 \\ \mathscr{L}_{j,i-1}(t) + L_{p_j(i)}\big(t + \mathscr{L}_{j,i-1}(t)\big) & \text{otherwise.} \end{cases} \tag{8.4}$$

The intuition of (8.4) is quite straightforward: the time it takes a packet to traverse the first i nodes is equal to the time required to traverse the first $i-1$ nodes plus the response-time of the i-th node. The *end-to-end response-time* of a flow j is therefore given by $\mathscr{L}_{j,\ell_j}(t)$, which we compactly denote by $\mathscr{L}_j(t)$.

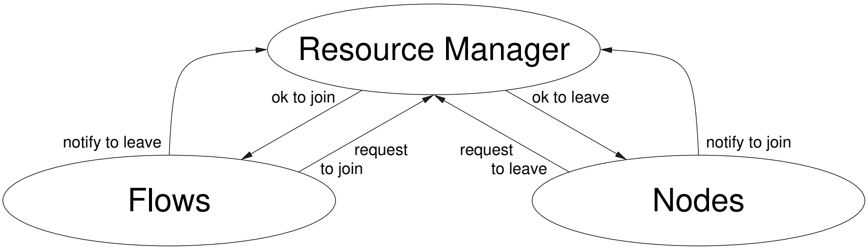


Figure 8.2 Scheme of interactions between the resource manager (which manages the network), the flows, and the nodes.

Model of the dynamic network

With the model of the flows $\mathscr{F}$ and the nodes $\mathscr{V}$ presented earlier, both nodes and flows are allowed to join and leave the network at run-time. Hence, we can finally define the network as follows:

DEFINITION 8.5—NETWORK

We define the *network* $\mathscr{G}(t)$ of the system at time t as the set of *nodes*, *directed edges*, and *flows* present at that time: $\mathscr{G}(t) = \{\mathscr{V}(t),\ \mathscr{E}(t),\ \mathscr{F}(t)\}$. □

Since we aim at guaranteeing end-to-end deadlines of the flows, the requests to join or leave the network must be properly handled by a *resource manager*, which manages the network (as illustrated in Figure 8.2). If not properly handled, the risk is that newly admitted flows may cause deadline-misses or that the uncontrolled departure of nodes may disconnect the network. The interactions between the resource manager and the flows are as follows:

- A flow $j \notin \mathscr{F}(t)$ may *request to join* the network. Such an instant is denoted by $f_j^{\mathsf{rq}+}$. When a new flow issues such a request, it also communicates to the resource manager the following information:

  1. its path p_j,
  2. its end-to-end deadline $\mathscr{D}_j$.

- After the request by a flow to join, the resource manager:

  1. accepts the flow j to the network at an instant $f_j^{\mathsf{ok}+}$ ($\geq f_j^{\mathsf{rq}+}$), if feasible, or
  2. rejects the flow j immediately, if not feasible.

  Details on the admission of a new flow based on its characteristics and the current state of the network are given in Section 8.3.

- A flow $j \in \mathscr{F}(t)$ may *notify and leave* the network at any time, and we denote this denote by f_j^-. In fact, it is only advantageous to let a flow (and its end-to-end deadline constraint) leave.
- The resource manager may notify a flow $j \in \mathscr{F}(t)$ that it has to leave the network. This might, for instance, happen if a node along the path p_j requests to leave the network. In such a case, the resource manager is no longer able to provide the requested services of the flow j.

The interactions between the resource manager and the nodes (which are the vertices of the graph) are as follows:

- A node $i \notin \mathscr{V}(t)$ may *notify and join* the network at any time. We denote such an instant by v_i^+. Since a node is bringing a new service to the network, there is no admission control to its request to join.
- A node $i \in \mathscr{V}(t)$ may *request to leave* the network to the resource manager. The instant of such a request is denoted by $v_i^{\mathsf{rq}-}$.
- The resource manager allows a node to leave only at time $v_i^{\mathsf{ok}-}$ $(\geq v_i^{\mathsf{rq}-})$. The time between $v_i^{\mathsf{rq}-}$ and $v_i^{\mathsf{ok}-}$ is needed by the resource manager to allow flows going through node i to leave the network in a graceful way.

Problem formulation The problem formulation in this chapter can now formally be summarized as follows;

- control the node deadlines $\mathbf{D}(t) = [D_i(t)]_{\forall i \in \mathscr{V}(t)}$,
- control when/how flows are allowed to join the network, and finally
- control when/how nodes are allowed to leave the network,

such that the end-to-end response times of all the flows in the network are always less than their end-to-end deadlines:

$$\forall t \geq 0,\ \forall j \in \mathscr{F}(t), \quad \mathscr{L}_j(t) \leq \mathscr{D}_j. \tag{8.5}$$

8.2 Static networks

In this section, we introduce a protocol which allows for *dynamic node deadlines* in *static networks*. By a static network we mean one where the network topology is constant, i.e., where no nodes or flows are allowed to join or leave the network, that is $\forall t \geq 0,\ \mathscr{G}(t) = \mathscr{G}$. Note that this still allow for the traffic rates and response times of nodes to change dynamically. This allows us to drop the time-dependency of the set of edges $\mathscr{E}(t) = \mathscr{E}$, the set of nodes $\mathscr{V}(t) = \mathscr{V}$, and the set of flows $\mathscr{F}(t) = \mathscr{F}$. However, we stress again that the node deadlines may change dynamically.

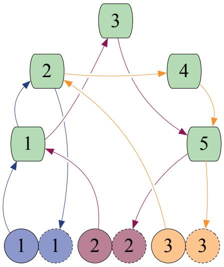


Figure 8.3 Example of a network. Nodes are represented by light yellow boxes. Flows are represented by: a source of packets (a colored circle labeled by the flow index), a path (a sequence of arrows from the source, through the nodes), and a destination of the packets (a colored circle with dashed boundary labeled by the flow index). This example of network is used to illustrate an issue with dynamic deadlines in Section 8.2.

To give some intuition of the challenges of guaranteeing end-to-end deadlines in a system with dynamic node deadlines (even for a static network), we begin by presenting a simple example. We then propose a solution, which allows for dynamic node deadlines and still provides guarantees on the end-to-end deadlines (which is formally proved in Theorem 8.1). Finally, we close this section by adapting the opening example and show that it is then able to guarantee that all the end-to-end deadlines are met for all times.

Example — issues with dynamic deadlines

If node deadlines were constant then a static assignment of node deadlines, equal to any vector of node deadlines $\mathbf{D} = \{D_i\}_{\forall i \in \mathscr{V}}$ satisfying

$$\mathbf{D} \in \overline{\mathbb{D}}(\mathscr{G}) = \left\{ D_i \in \mathbb{R}^+ : \forall i \in \mathscr{V},\ D_i \geq \underline{L}_i, \quad \forall j \in \mathscr{F},\ \sum_{i=1}^{\ell_j} D_{p_j(i)} \leq \mathscr{D}_j \right\}, \tag{8.6}$$

would guarantee that no end-to-end deadlines would be missed. Equation (8.6) is very similar to the constraints presented in Chapter 7, Equation (7.2), but with the addition of the lower bound $\underline{L}_i$. The intuition behind Equation (8.6) is simple: the sum of the node deadlines along the path p_j of a flow j cannot exceed the end-to-end deadline $\mathscr{D}_j$ of the path j.

We remark that the set $\overline{\mathbb{D}}(\mathscr{G})$ is convex, since it is the polytope built from the intersection among linear half-spaces. This also implies that every line between any two points in $\overline{\mathbb{D}}(\mathscr{G})$ also belongs to $\overline{\mathbb{D}}(\mathscr{G})$.

Let us now consider the issues of performing a transition from some deadline assignment $\mathbf{D}(t_1) \in \overline{\mathbb{D}}(\mathscr{G})$ to another assignment $\mathbf{D}(t_2) \in \overline{\mathbb{D}}(\mathscr{G})$, hence with both the starting and ending node deadlines belonging to $\overline{\mathbb{D}}(\mathscr{G})$. Since $\overline{\mathbb{D}}(\mathscr{G})$ is convex then

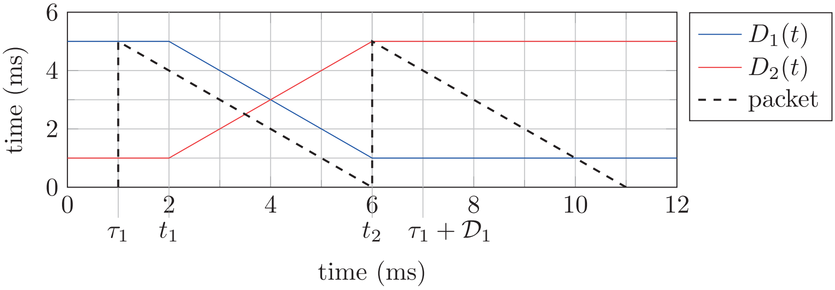


Figure 8.4 Example of how dynamic node deadlines may lead to end-to-end deadline violations. The node deadlines $D_1(t)$ (in blue) and $D_2(t)$ (in red) are changed from $\mathbf{D}(t_1) = [5,1,\ldots]$ to $\mathbf{D}(t_2) = [1,5,\ldots]$. This may cause a packet in flow 1 (with path $p_1 = \{1,2\}$, as shown in Figure 8.3) to miss its end-to-end deadline $\mathscr{D}_1 = 6$.

the linear transition of node deadlines

$$\mathbf{D}(t) = \mathbf{D}(t_1) \times \frac{t_2 - t}{t_2 - t_1} + \mathbf{D}(t_2) \times \frac{t - t_1}{t_2 - t_1}$$

always belongs to $\overline{\mathbb{D}}(\mathscr{G})$ for all $t \in [t_1, t_2]$.

However, even if $\forall t \in [t_1, t_2]$ the linear combination $\mathbf{D}(t)$ always belongs to $\overline{\mathbb{D}}(\mathscr{G})$, the end-to-end deadline may be missed anyway. Suppose we have the network $\mathscr{G}$, illustrated in Figure 8.3 and focus on flow 1 (the blue flow), with path $p_1 = \{1,2\}$. Assume that the end-to-end deadline for this flow is $\mathscr{D}_1 = 6$ ms. Suppose also that in response to an increase of the incoming packets rate, node 2 must change its deadline from $D_2(t_1) = 1$ to $D_2(t_2) = 5$. The node deadlines of the system are therefore changing from $\mathbf{D}(t_1) = [5,\ 1,\ \ldots]$ to $\mathbf{D}(t_2) = [1,\ 5,\ \ldots]$. Please, note that for the purpose of this example the particular choice of deadlines for nodes 3, 4, and 5 does not matter.

Figure 8.4 shows the deadlines of node 1 and node 2 over time. At time $\tau_1 = 1$ a packet enters node 1, which has $D_1(\tau_1) = 5$. In the worst case, node 1 will have a response-time of $R_1(t) = D_1(t)$, which means it will finish processing that packet at time $\tau_1 + D_1(\tau_1) = 6$ms.

Suppose now, that at time $t_1 = 2$ms, while this packet is still at node 1, the resource manager begins changing the node deadlines from $\mathbf{D}(t_1) = [5,1,\ldots]$ towards $\mathbf{D}(t_2) = [1,5,\ldots]$. When the packet exits node 1 at time $\tau_1 + D_1(\tau_1) = t_2 = 6$, then it enters node 2. Due to the change of node deadlines, now the value of $D_2(6) = 6$. This means that, in the worst case (if $R_2(t) = D_2(t)$), the response time of node 2 is also 5 ms.

The packet that entered flow 1 at time $\tau_1 = 1$ would therefore, in the worst case, take 10 ms to traverse its path p_1, hence, violating its end-to-end deadline $\mathscr{D}_1 = 6$.

The reason for this is that with dynamic node deadlines:

$$\mathscr{R}_1(\tau_1) \neq D_1(\tau_1) + D_2(\tau_1),$$

but instead

$$\mathscr{R}_1(\tau_1) = D_1(\tau_1) + D_2\big(\tau_1 + D_1(\tau_1)\big) = D_1(\tau_1) + D_2(\tau_1) + \int\limits_{\tau_1}^{\tau_1+D_1(\tau_1)} \dot{D}_2(x)\mathrm{d}x.$$

This simple example shows that when the network is allowed to change the node deadlines dynamically, $\mathbf{D}(t) \in \overline{\mathbb{D}}(\mathscr{G})$ is not a sufficient condition to guarantee end-to-end deadlines to be met. The node deadlines $\mathbf{D}(t)$ thus need to satisfy a stricter constraint, which is discussed next.

Guaranteeing end-to-end deadlines

In this section, we provide the conditions that allow the network to dynamically change the node deadlines without incurring any end-to-end violations, as illustrated in the previous example. In Protocol 1 we present a solution to this problem. The intuition is that by limiting the rate-of-change of the node deadlines, it is possible to compute how the sum of the node deadlines along a path p_j will change over time. This in turn, allows us to upper-bound the end-to-end response-time of the flow. By then choosing the node deadlines in a clever way, we can guarantee that the upper-bound of the end-to-end response-time never exceeds the end-to-end deadline. This is formally proved in Theorem 8.1.

Protocol 1 Management of dynamic node deadlines in static networks

- The node deadlines $D_i(t)$ can never change with a rate larger than some fixed constant $\alpha \in [0,1]$:

$$\forall t \geq 0,\ \forall i \in \mathscr{V},\ |\dot{D}_i(t)| \leq \alpha.$$

- The node deadlines must always remain within the *safe space* $\mathbb{D}(\mathscr{G})$:

$$\mathbf{D}(t) \in \mathbb{D}(\mathscr{G}) = \Big\{ D_i \in \mathbb{R}^+ \ :\ i \in \mathscr{V},\ \underline{L}_i \leq D_i,\ \forall j \in \mathscr{F},\ \sum_{i=1}^{\ell_j} (1+\alpha)^{\ell_j - i} D_{p_j(i)} \leq \mathscr{D}_j \Big\}.$$

THEOREM 8.1—DYNAMIC DEADLINES IN STATIC NETWORKS
No end-to-end deadline of any flow is violated, that is

$$\forall t \geq 0,\ \forall j \in \mathscr{F}, \qquad \mathscr{L}_j(t) \leq \mathscr{D}_j, \tag{8.7}$$

as long as the node deadlines $\mathbf{D}(t)$ never change with a rate faster than some fixed, given bound $\alpha \in [0,1]$:

$$\forall t \geq 0,\ \forall i \in \mathscr{V}, \qquad |\dot{D}_i(t)| \leq \alpha, \tag{8.8}$$

and as long as the node deadlines remain within the safe space:

$$\forall t \geq 0, \qquad \mathbf{D}(t) \in \mathbb{D}(\mathscr{G}), \tag{8.9}$$

with $\mathbb{D}(\mathscr{G})$ given by

$$\mathbb{D}(\mathscr{G}) = \Big\{ D_i \in \mathbb{R}^+ : \forall i \in \mathscr{V},\ \underline{L}_i \leq D_i, \quad \forall j \in \mathscr{F},\ \sum_{i=1}^{\ell_j} (1+\alpha)^{\ell_j - i} D_{p_j(i)} \leq \mathscr{D}_j \Big\}. \tag{8.10}$$

□

Proof We begin by recalling Assumption 8.1: the nodes in the network have a response-time controller which ensures that Equation (8.1) always holds, that is $\forall t \geq 0,\ \underline{L}_i \leq L_i(t) \leq D_i(t)$. It therefore follows that for the first node $p_j(1)$ of any path $j \in \mathscr{F}$, it is always true that:

$$\forall t \geq 0,\ \forall j \in \mathscr{F}, \quad \mathscr{L}_{j,1}(t) = L_{p_j(1)}(t) \leq D_{p_j(1)}(t). \tag{8.11}$$

It then follows from Equation (8.4), that for all subsequent nodes on the path p_j with index $i = 2, \ldots, \ell_j$ it holds that:

$$\begin{aligned} \forall t \geq 0,\ \forall j \in \mathscr{F}, \quad \mathscr{L}_{j,i}(t) &= \mathscr{L}_{j,i-1}(t) + L_{p_j(i)}\big(t + \mathscr{L}_{j,i-1}(t)\big) \\ &\leq \mathscr{L}_{j,i-1}(t) + D_{p_j(i)}\big(t + \mathscr{L}_{j,i-1}(t)\big) \\ &\leq \mathscr{L}_{j,i-1}(t) + D_{p_j(i)}(t) + \alpha\, \mathscr{L}_{j,i-1}(t) \\ &= (1+\alpha)\mathscr{L}_{j,i-1}(t) + D_{p_j(i)}(t) \end{aligned} \tag{8.12}$$

where each step follows:

1. from the definition of end-to-end response-time (8.4),
2. from Equation (8.1),
3. from Equation (8.8), which implies Lipschitz-continuity of $D_i(t)$ with Lipschitz-constant α,
4. from basic math.

In the next step, we show that

$$\forall t \geq 0,\ \forall j \in \mathscr{F}, \quad \forall x = 1, \ldots, \ell_j, \qquad \mathscr{L}_{j,x}(t) \leq \sum_{i=1}^{x} (1+\alpha)^{x-i} D_{p_j(i)}(t), \quad (8.13)$$

holds by proving by induction on the index x of nodes over the path j. When $x = 1$, Equation (8.13) follows directly from (8.11). For any other $x > 1$, we have that

$$\begin{aligned} \mathscr{L}_{j,x}(t) &\leq (1+\alpha)\mathscr{L}_{j,x-1}(t) + D_{p_j(x)}(t) \\ &\leq (1+\alpha)\sum_{i=1}^{x-1}(1+\alpha)^{x-1-i} D_{p_j(i)}(t) + D_{p_j(x)}(t) \\ &= \sum_{i=1}^{x-1}(1+\alpha)^{x-i} D_{p_j(i)}(t) + D_{p_j(x)}(t) \\ &= \sum_{i=1}^{x}(1+\alpha)^{x-i} D_{p_j(i)}(t) \end{aligned}$$

where the different steps follow:

1. because the inequality is the same as Equation (8.12)
2. because we exploit the inductive hypothesis of (8.13) for $x-1$,
3. from basic math.

Hence, Equation (8.13) is proved for all $x = 1, \ldots, \ell_j$.

Finally, we can conclude the proof by showing that

$$\forall t \geq 0,\ \forall j \in \mathscr{F} \qquad \mathscr{L}_j(t) = \mathscr{L}_{j,\ell_j}(t) \leq \sum_{i=1}^{\ell_j}(1+\alpha)^{\ell_j - i} D_{p_j(i)}(t) \leq \mathscr{D}_j$$

which implies that as long as the node deadlines are chosen such that $\mathbf{D}(t) \in \mathbf{D}(\mathscr{G})$ no end-to-end deadlines of any flow $j \in \mathscr{F}$ is violated, and the theorem is proved.□

Example — fixed by applying Theorem 8.1

This section illustrates that if the node deadlines are changed in accordance to Protocol 1, then the issue of end-to-end deadline misses shown in the previous example cannot happen. To do so, we modify the previous example but still consider the flow 1 of Figure 8.3, which has path $p_1 = \{1,2\}$ and end-to-end deadline $\mathscr{D}_1 = 6$. We also assume that the resource manager must change $D_2(t)$ from $D_2(t_1) = 1$ to $D_2(t_2) = 5$.

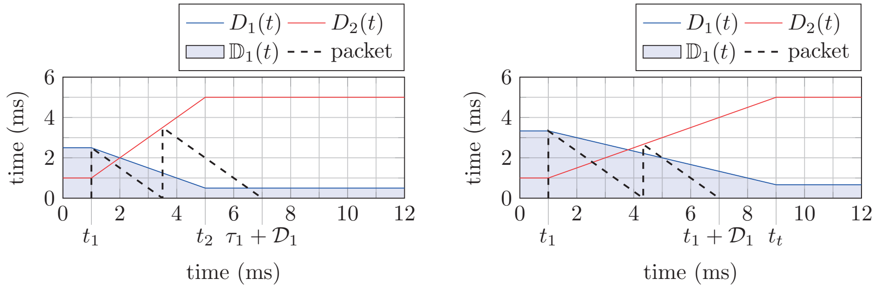
Recalling the previous example we can now verify that if the node deadlines begin in a state of $D_1(t_1) = 5$ and $D_2(t_1) = 1$ the resource manager is not able to change them at all. In fact, by writing explicitly the constraint of Equation (8.10) for the path $j = 1$, we have

$$\begin{aligned} &(1+\alpha)D_1(t_1)+D_2(t_1) \leq \mathscr{D}_1 \\ &(1+\alpha)1+5 \leq 6 \quad \Rightarrow \quad \alpha \leq 0 \end{aligned}$$

which means that node deadlines are not allowed to change at all. If some change is needed (for example, to handle a burst of incoming traffic), then the node deadlines must be more constrained. In fact, given the change of $D_2(t)$, the choice of $D_1(t)$ must satisfy the following constraint:

$$\forall t \geq 0, \quad (1+\alpha)D_1(t)+D_2(t) \leq \mathscr{D}_1 = 6. \tag{8.14}$$

Next, we compare two cases of different values of α to see how it affects how quickly one can change the node deadlines, but also how constrained the possible choices of $D_1(t)$ becomes. The results of these examples can be seen in Figure 8.5.



(a) Maximum rate-of-change $\alpha = 1$

(b) Maximum rate-of-change $\alpha = 0.5$

Figure 8.5 Modified version of the previous example. We illustrate how the system dynamically changes node deadlines over time, with different rates of change. In 8.5(a) and 8.5(b) the nodes deadlines are allowed to change with a rate of $\alpha = 1$ and $\alpha = 0.5$, respectively. Given the change of $D_2(t)$ (red line) from 1 to 5, the resource manager is only allowed to change $D_1(t)$ (blue line) ensuring that it remains within the safe space $\mathbb{D}_1(t)$ (shaded blue area). Finally, as illustrated by the dashed black line, by ensuring that $D_1(t)$ remains within the blue region, the longest time it will take a packet to traverse flow 1 will be less than its end-to-end deadline.

Maximum rate-of-change $\alpha = 1$ To allow the network to transition as quickly as possible from $D_2(t_1) = 1$ to $D_2(t_2) = 5$ we choose $\alpha = 1$. This means that the choices of D_1 are constrained by the following condition:

$$\forall t \geq 0, \quad (1+1)D_1(t)+D_2(t) \leq \mathscr{D}_1 = 6.$$

This condition gives the resource manager the safe space for $D_1(t)$ illustrated by the shaded blue region in Figure 8.5(a). The largest possible choice for $D_1(t)$ is therefore to change it from $D_1(t_1) = 2.5$ to $D_1(t_2) = 0.5$, as illustrated by the blue line in Figure 8.5(a).

Maximum rate-of-change $\alpha = 0.5$ The stringency of the constraint on the node deadlines when $\alpha = 1$ may be relaxed by requiring a slower transition between the node deadlines, for example $\alpha = 0.5$. By doing so, we get the following conditions on $D_1(t)$

$$D_1(t_1) \leq \frac{10}{3} \approx 3.333, \qquad D_1(t_2) \leq \frac{2}{3} \approx 0.666,$$

in order to ensure that Equation (8.14) holds, and assuming that $D_2(t)$ is again changed from $D_2(t_1) = 1$ to $D_2(t_2) = 5$. Not surprisingly, by requiring a smoother transition, the safe space $\mathbb{D}_1(t)$ (shaded blue area in Figure 8.5(b)) becomes larger. Choosing $D_1(t)$ to be as large as possible, thus means changing $D_1(t)$ from $D_1(t_1) = 10/3$ to $D_1(t_2) = 2/3$, as illustrated in the figure.

Trade-offs with alpha These two examples illustrate some fundamental trade-offs that come by adopting Protocol 1. While it allows the system to change the node deadlines dynamically with a rate of α, it imposes some constraints on the possible choices of node deadlines. The quicker one wishes to change the node deadlines, the more restricted the choice of feasible node deadlines becomes, and vice verse The design-parameter α should be chosen appropriately for a given application.

8.3 Dynamic networks

In this section, we generalize the methodology presented in Section 8.2 to the case of a *dynamic network* $\mathscr{G}(t) = \{\mathscr{V}(t),\ \mathscr{E}(t),\ \mathscr{F}(t)\}$, where nodes and flows may join and leave the network at run-time. We begin by showing, in Corollary 8.1, that the result of Theorem 8.1 transfers directly to a dynamic network. This means that the end-to-end deadline of any flow in the network will be met as long as the hypothesis of Theorem 8.1 hold for all states of the network $\mathscr{G}(t)$ and at all times, and as long as the conditions stated in Corollary 8.1 are fulfilled. By comparing Theorem 8.1 and Corollary 8.1, one can see that the only difference is that we now allow for a dynamic network, i.e., $\mathscr{G}(t)$ instead of a static network $\mathscr{G}$.

COROLLARY 8.1—DYNAMIC DEADLINES IN DYNAMIC NETWORKS
The end-to-end deadline of all flows are always met, at all times, that is:

$$\forall t \geq 0,\ \forall j \in \mathscr{F}(t) \quad \mathscr{L}_j(t) \leq \mathscr{D}_j, \tag{8.15}$$

as long as the node deadlines $\mathbf{D}(t)$ never change with a rate faster than some fixed bound $\alpha \in [0,1]$:

$$\forall t \geq 0,\ \forall i \in \mathscr{V}(t), \qquad |\dot{D}_i(t)| \leq \alpha, \tag{8.16}$$

and as long as the node deadlines always remain within the safe space $\mathbb{D}\big(\mathscr{G}(t)\big)$:

$$\forall t \geq 0, \quad \mathbf{D}(t) \in \mathbb{D}\big(\mathscr{G}(t)\big), \tag{8.17}$$

with $\mathbb{D}(\mathscr{G}(t))$ given by

$$\mathbb{D}\big(\mathscr{G}(t)\big) = \Big\{ D_i \in \mathbb{R}^+ : \forall i \in \mathscr{V}(t),\, \underline{L}_i \leq D_i, \quad \forall j \in \mathscr{F}(t),\, \sum_{i=1}^{\ell_j} (1+\alpha)^{\ell_j - i} D_{p_j(i)} \leq \mathscr{D}_j \Big\}. \tag{8.18}$$

□

Proof We begin the proof by observing that from Equation (8.16), it follows directly from Theorem 8.1 that for a fixed time-instance τ it holds that

$$\forall j \in \mathscr{F}(\tau), \quad \mathscr{L}_j(\tau) \leq \mathscr{D}_j, \tag{8.19}$$

as long as

$$\mathbf{D}(\tau) \in \mathbb{D}\big(\mathscr{G}(\tau)\big), \tag{8.20}$$

with $\mathbb{D}\big(\mathscr{G}(\tau)\big)$ given by Equation (8.18). In fact, this is precisely what was stated and proved in Theorem 8.1, but with a fixed network topology $\mathscr{G}$, instead of an instantaneous “snapshot” $\mathscr{G}(\tau)$ of a dynamic topology.

The hypothesis of Equation (8.17) then ensures that Equation (8.20) holds $\forall t \geq 0$. Therefore, it follows that Equation (8.19) holds $\forall t \geq 0$, and in turn that Equation (8.15) also holds $\forall t \geq 0$, as required. □

As demonstrated by the short proof, Corollary 8.1 does not pose any deeper conceptual challenges compared to Theorem 8.1. However, the two hypotheses of (8.16) and (8.17) may be hard to enforce simultaneously, if no special care is taken. We illustrate this through the next example.

Example — issues in acceptance of a new flow The blind admission of new flows as soon as they request to join, may cause the violation of one of the two hypothesis of the corollary (Equations (8.16) and (8.17)) making Corollary 8.1 incapable of guaranteeing the end-to-end deadlines. At the time when any new flow j' is admitted to the network, the set of flows $\mathscr{F}(t)$ includes the new flow j' which was not previously in the set. As a consequence of the acceptance of the new flow j' into $\mathscr{F}(t)$, the safe space $\mathbb{D}\big(\mathscr{G}(t)\big)$ may suddenly shrink due to the newly added constraint. As illustrated in the next example, this might in turn cause the node deadlines to be in an infeasible state, such that the end-to-end deadline of the newly accepted flow will be violated. Notice that node deadlines **cannot** instantaneously adapt to the new constraint, otherwise the hypothesis of “bounded rate of change” of Equation (8.16) is violated.

In Figure 8.6, we illustrate a system where a new flow j' registers to join the network at time $f_{j'}^{\mathsf{rq}+} = 3$. As soon as this flow is accepted into the network, at time $f_{j'}^{\mathsf{ok}+} = 4.5$, the set of feasible node deadlines (illustrated by the shaded green area) make a discrete jump. This means that the node deadlines $\mathbf{D}(t)$ (black thick line in Figure 8.6) will no longer remain within $\mathbb{D}\big(\mathscr{G}(t)\big)$. In other words this means that $\mathbf{D}(f_{j'}^{\mathsf{ok}+}) \notin \mathbb{D}\big(\mathscr{G}(f_{j'}^{\mathsf{ok}+})\big)$, which is a clear violation of the conditions of Corollary 8.1.

The illustrated issue suggests that special care must be taken when the network $\mathscr{G}(t)$ is modified, since the discrete variation of the network may not be compatible with the need of smooth node deadlines. Therefore, we present two new protocols.

Protocol allowing dynamic networks

Next, we present two new protocols:

- Protocol 2 explains how the requests of flows are managed, while
- Protocol 3 illustrates the management of nodes of the network.

We show that by following these protocol, the hypothesis of Corollary 8.1 always holds, making the corollary applicable.

Management of flows The intuition behind Protocol 2, which manages the flows, comes from the fact that as soon as a new flow j' is accepted into the network, at time $f_{j'}^{\mathsf{ok}+}$, the network changes from $\mathscr{G} = \mathscr{G}(t) = \{\mathscr{V}(t),\ \mathscr{E}(t),\ \mathscr{F}(t)\}$ into $\mathscr{G}^+ = \mathscr{G}(f_{j'}^{\mathsf{ok}+}) = \big\{\mathscr{V}(t),\ \mathscr{E}(t),\ \mathscr{F}(t) \cup \{j'\}\big\}$. The constraint corresponding to the new flow j' is thus added to $\mathbb{D}(\mathscr{G})$ leading to the new set of feasible node deadlines $\mathbb{D}(\mathscr{G}^+) \subseteq \mathbb{D}(\mathscr{G})$. Therefore, in order to ensure that the node deadlines remain within the safe space once j' is accepted, i.e., that $\mathbf{D}(f_{j'}^{\mathsf{ok}+}) \in \mathbb{D}\big(\mathscr{G}(f_{j'}^{\mathsf{ok}+})\big)$, Protocol 2 will

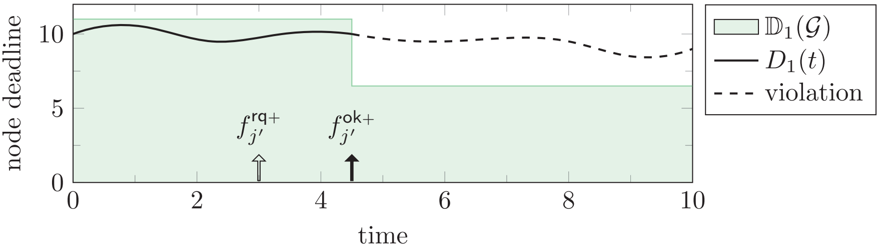


Figure 8.6 Illustration of why it might be difficult to ensure that $\forall t \geq 0,\ \mathbf{D}(t) \in \mathbb{D}\big(\mathscr{G}(t)\big)$ and $|\dot{D}_i(t)| \leq \alpha$. In this scenario, the safe space for the first node, $\mathbb{D}_1\big(\mathscr{G}(t)\big)$ (shaded green area), makes a discrete jump as soon as the new flow j' is accepted into the network. The reason is that the inclusion of the new flow adds a new end-to-end deadline constraints. This, in turn, may cause the node deadlines to instantaneously become infeasible, “leaving” the safe space (illustrated by the dashed line).

only accept j' as long as $\mathbf{D}(t) \in \mathbb{D}(\mathscr{G}^+)$. One should node that, if this is already the case when request of flow j' is made (at time $t = f_{j'}^{\mathsf{rq}+}$), it will be allowed to join immediately:

$$\text{if } \mathbf{D}(f_{j'}^{\mathsf{rq}+}) \in \mathbb{D}(\mathscr{G}^+) \text{ then } f_{j'}^{\mathsf{ok}+} = f_{j'}^{\mathsf{rq}+}.$$

If on the other hand, $\mathbf{D}(f_{j'}^{\mathsf{rq}+}) \notin \mathbb{D}(\mathscr{G}^+)$, it is not possible to accept j' immediately. Therefore, in order to accept it, the resource manager changes the node deadlines towards a *goal point within the new safe space* $\mathbf{D}^* \in \mathbb{D}(\mathscr{G}^+)$. It will then accept the new flow j', once $\mathbf{D}(t) = \mathbf{D}^*$, which will occur at time $f_{j'}^{\mathsf{ok}+}$, given by Equation (8.21).

Protocol 2 Management of flows

- At time $f_{j'}^{\mathsf{rq}+}$, the new flow j' requests to join
- If $\mathbf{D}(f_{j'}^{\mathsf{rq}+}) \in \mathbb{D}(\mathscr{G}^+)$, then the flow j' is admitted immediately, that is $f_{j'}^{\mathsf{ok}+} = f_{j'}^{\mathsf{rq}+}$.
- If $\mathbf{D}(f_{j'}^{\mathsf{rq}+}) \notin \mathbb{D}(\mathscr{G}^+)$, then the admission of flow j' is delayed until the node deadlines have completed a linear transition to a *goal point* $\mathbf{D}^* \in \mathbb{D}(\mathscr{G}^+)$, that is
$$f_{j'}^{\mathsf{ok}+} = f_{j'}^{\mathsf{rq}+} + \frac{\max_{i \in \mathscr{V}(t)} \left| D_i^* - D_i(f_{j'}^{\mathsf{rq}+}) \right|}{\alpha}. \tag{8.21}$$
with α being the maximum feasible rate of change of node deadlines.
- If $\mathbf{D}\big(\mathscr{G}^+)\big) = \varnothing$, then the request of flow j' to join the network is rejected.
- A flow $j \in \mathscr{F}(t)$ may notify the resource manager and leave the network at any time. The time when it leaves is denoted by f_j^-.

Let us comment on the choice of the “goal point” $\mathbf{D}^*$. In general, any choice of $\mathbf{D}^* \in \mathbb{D}(\mathscr{G}^+)$ is valid. However, if the target is to minimize the transition-time from the time of the request to join to the time of acceptance, then it should be chosen as

$$\mathbf{D}^* = \arg\min_{\mathbf{D} \in \mathscr{G}^+} \max_{i \in \mathscr{V}(t)} \left| D_i - D_i(f_{j'}^{\mathsf{rq}+}) \right|.$$

We would like to point out three observations from Protocol 2. The first one is that during the transition to accept the new flow j' the node deadlines are changed according to the following linear function:

$$\forall i \in \mathscr{V}(f_{j'}^{\mathsf{rq}+}),\ \forall t \in [f_{j'}^{\mathsf{rq}+}, f_{j'}^{\mathsf{ok}+}], \quad D_i(t) = \frac{D_i(f_{j'}^{\mathsf{rq}+}) \times (t - f_{j'}^{\mathsf{rq}+})}{f_{j'}^{\mathsf{ok}+} - f_{j'}^{\mathsf{rq}+}} + \frac{D_i^* \times (f_{j'}^{\mathsf{ok}+} - t)}{f_{j'}^{\mathsf{ok}+} - f_{j'}^{\mathsf{rq}+}}. \tag{8.22}$$

This means that the node deadlines are changed along a line from $\mathbf{D}(f_{j'}^{\mathsf{rq}+})$ to $\mathbf{D}(f_{j'}^{\mathsf{ok}+})$. Since $\mathbb{D}(\mathscr{G})$ is a convex space, it follows that $\mathbf{D}(t) \in \mathbb{D}(\mathscr{G})$ during the transition. Hence, it also follows that the hypothesis of Equation (8.17) in Corollary 8.1 holds during the transition.

The second observation is that by changing the node deadlines according to Equation (8.22) we acquire the property that $D_i(f_{j'}^{\mathsf{ok}+}) = D_i^*$. This means that at the end of the transition, at time $t = f_{j'}^{\mathsf{ok}+}$, we have $\mathbf{D}(t) \in \mathbb{D}(\mathscr{G}^+)$. This implies that once the new flow j' is accepted, condition (8.17) of Corollary 8.1 continues to hold.

The final observation is that by exploiting the value of $f_{j'}^{\mathsf{ok}+}$ of Equation (8.21), we have

$$|\dot{D}_i(t)| = \frac{|D_i(f_{j'}^{\mathsf{rq}+}) - D_i^*|}{f_{j'}^{\mathsf{ok}+} - f_{j'}^{\mathsf{rq}+}} = \alpha \, \frac{|D_i(f_{j'}^{\mathsf{rq}+}) - D_i^*|}{\max_{i \in \mathscr{V}(t)} \left| D_i^* - D_i(f_{j'}^{\mathsf{rq}+}) \right|} \leq \alpha \tag{8.23}$$

This implies that Protocol 2 fulfills condition (8.16) of Corollary 8.1, since the maximum rate-of-change of any node is α during the transition.

By combining all three observations, we can conclude that Protocol 2 ensures that the hypotheses of Corollary 8.1 are never violated while accepting new flows, and therefore that no end-to-end deadlines will be guaranteed.

Finally, we remark that if there is no possible choice of a goal point, i.e., if $\mathbb{D}(\mathscr{G}^+) = \varnothing$, then the request of j' to join is clearly rejected because the admission of the new flow j' may cause the violation of end-to-end deadline of some flow already admitted to the network.

Management of nodes The basic idea behind Protocol 3 is that when a node $i \in \mathscr{V}(t)$ leaves the network, it affects all the flows having a path that is passing through it. Therefore, at time $v_i^{\mathsf{rq}-}$, that is when node i requests to leave the network, the resource manager notifies all the affected flows $j \in \{j \,:\, \forall j \in \mathscr{F}(t),\ i \in p_j\}$ that it will be pushed out from the network after a time T_i^{leave}. The rationale for this is simply that the resource manager will no longer be able to provide any guarantees for the end-to-end deadlines of the affected flows once node i has left the network.

However, should the affected flows still wish to use some of the services in the network, they may request to re-join the network as a new flow. In order to allow the affected flows adequate time to do this, and to also ensure that there is enough time for the packets currently in the affected flows, we require T_i^{leave} to be greater than the largest end-to-end deadline of the affected flows.

Moreover, it can be noticed that there is no condition on the nodes willing to join the network, thus any node wanting to join the network can do so immediately.

Protocol 3 Management of nodes

- A node $i \notin \mathscr{V}(t)$ may notify the resource manager and join the network at any time v_i^+.
- When a node $i \in \mathscr{V}(t)$ requests to leave the network (the time of which is denoted by $v_i^{\mathsf{rq}-}$), it is allowed to do so at time $v_i^{\mathsf{ok}-}$:

  $$v_i^{\mathsf{ok}-} \geq v_i^{\mathsf{rq}-} + T_i^{\text{leave}}$$

  with $T_i^{\text{leave}} \geq \max\{\mathscr{D}_j \,:\, \forall j \in \mathscr{F}(t),\, i \in p_j\}$
- The resource manager will notify all the affected flows

  $$j \in \{j \,:\, \forall j \in \mathscr{F}(t),\, i \in p_j\}$$

  that they will be pushed out at time $t = v_i^{\mathsf{ok}-}$ if they have not left the network by then.

Comments on handling multiple flows It should be noted that while Protocol 2 only treats the case where a single flow j' requests to join, it is possible to allow multiple flows to request. The management of this scenario can be achieved by introducing a *request queue*, and then applying Protocol 2 for the request at the head of the queue. Once the request is served, the resource manager can then repeat for the new head-of-the-queue request until there are no more pending requests. This methodology would only incur in a heavier notation which we prefer not to add to lighten the presentation.

Another option would be to batch many requests together, and then apply Protocol 2 on the new batch of flows. This approach could naturally also be combined with the request queue for new requests.

Later, in Section 8.4 we illustrate through simulations that new flows can be accepted in matter of milliseconds, or even micro seconds. Therefore, given the application of cloud robotics, it is fair to assume that there will not be multiple flows requesting to join simultaneously. And should there be, introducing a request queue will not incur any significant delay for accepting new flow.

Example — dynamic network topology

In this subsection, by using some examples, we illustrate how the two new protocols work. The scenario is illustrated in Figure 8.7 and consists of the two transitions. In the first transition, from t_1 to t_2, flow 4 (green) requests to join the network. The second transition, from t_2 to t_3 illustrates how node 6 requests to leave the network. By doing so, it will affect flow 3, which has a path passing through node $i = 6$. The affected flow must therefore leave the network, and re-join as a new flow

$j = 6$, illustrated by the gray flow in Figure 8.7. The schedule and time-line for both transitions are illustrated in Figure 8.8.

Request of a flow to join When the new flow 4 requests to join the network, at time $f_4^{\mathsf{rq}+} = 3$, the node deadlines of the system are in a state such that it cannot be accepted immediately, i.e., $\mathbf{D}(f_4^{\mathsf{rq}+}) \notin \mathbb{D}(\mathscr{G}^+)$. According to Protocol 2, this requires the resource manager to change the node deadlines to a goal point $\mathbf{D}^* \in \mathbb{D}(\mathscr{G}^+)$. The node deadlines will therefore be changed linearly from $\mathbf{D}(f_4^{\mathsf{rq}+})$ to $\mathbf{D}^*$. At time $f_4^{\mathsf{ok}+}$, we have that $\mathbf{D}(t) = \mathbf{D}^*$, and then flow 4 is admitted into the network. This is illustrated in Figure 8.7 with the safe space for the current network $\mathbb{D}(\mathscr{G})$ shown as the shaded green region, and the safe space for the new network $\mathbb{D}(\mathscr{G}^+)$ as the shaded blue region, and finally the goal point $\mathbf{D}^*$ as the $*$ symbol.

Request of a node to leave At time $v_6^{\mathsf{rq}-} = 9$, node 6 requests to leave the network, as illustrated in Figure 8.8. Since node 6 belongs to the path p_3 of flow 3 (see the yellow flow in Figure 8.7), the departure of node 6 would affect flow 3, with end-to-end deadline $\mathscr{D}_3 = 5$. By following Protocol 3, the resource manager will therefore notify flow 3 that it will no longer provide any end-to-end deadline guarantees after time

$$v_6^{\mathsf{ok}-} = v_6^{\mathsf{rq}-} + T_6^{\mathrm{leave}} = 9 + 5 = 14.$$

The node will then be allowed to leave the network at this time $v_6^{\mathsf{ok}-}$, as illustrated in Figure 8.8. In the figure, it can also be noticed that the safe space increases when node 6 leaves. The reason is that constraint of the end-to-end deadline $\mathscr{D}_3$ of flow 3 is removed as soon as this flow leaves the network.

In this example, we assume that flow 3 still wants to remain in the network. Therefore, it will request to re-join the network as a new flow 5 at time $f_5^{\mathsf{rq}+}$. At this time, the node deadlines already allow the requesting flow 5 to join, i.e., that

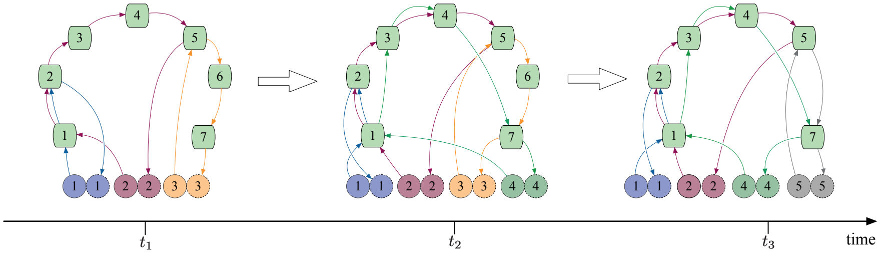


Figure 8.7 Illustration of how the network changes in the example of Section 8.3. In the first transition, from time t_1 to t_2, flow, 4 (green), joins the network, which already has flows 1, 2, and 3. Then in the second transition, from time t_2 to t_3, node 6 leaves the network. When node 6 leaves the network, it affects flow 3 (yellow), which then leaves, and re-joins the network as a new flow 5 (gray).

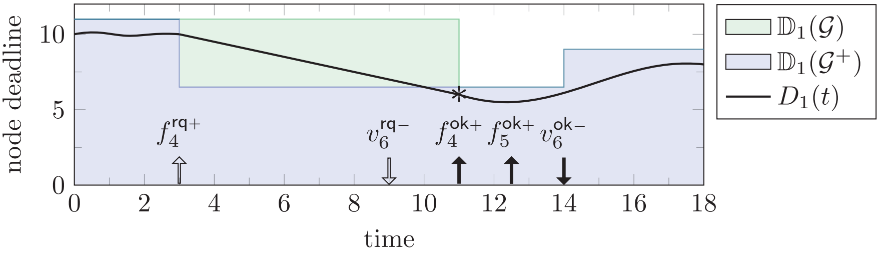


Figure 8.8 Illustration of how the space of feasible node deadlines changes when new flows are accepted into the network, as well as when nodes leave. It shows a request from flow $j' = 4$ to join the network at time $f_4^{\mathsf{rq}+} = 3$ as well as a request from node $i = 6$ to leave the network at time $v_6^{\mathsf{rq}-} = 9$. It also illustrates how the resource manager changes the node deadlines towards $\mathbf{D}^*$ (given by $*$) before accepting the new flow $j' = 4$.

$\mathbf{D}(f_5^{\mathsf{rq}+}) \in \mathbb{D}(\mathscr{G}^+)$, and then flow 5 is admitted immediately ($f_5^{\mathsf{ok}+} = f_5^{\mathsf{rq}+}$), as shown in Figure 8.8.

8.4 Evaluation: trade-offs with alpha

In this section we evaluate some of the effects introducing Protocols 1, 2, and 3 might have on a system. We are particularly concerned with the trade-offs of the design-parameter α. The intuition is that by choosing a value for α we choose how quickly the resource manager is able to change the node deadlines in the network. A higher value of α will therefore allow for quicker changes. This will allow the resource manager to accept new flows faster. However, as illustrated in Section 8.2, a higher value of α will also require lower node deadlines. This means that the response-times of the nodes in the network have to be lower. Should the response-time controller be using the AutoSAC-methodology, then this might lead to more packets being discarded, as was investigated in Chapter 6 when the ratio between Δ_i/D_i and its affect on performance was treated.

System used for evaluation To evaluate the trade-offs of α, we will use the system presented in Chapter 7. The network topology used for the evaluation is given by Figure 8.9 and is given by 5 nodes and 3 flows. The difference from the system of the previous chapter, however, is that we will now augment the optimization problem for assigning node deadlines (7.11), with the added constraints of (8.17). Moreover, we also assumed that every node in the network has the same time-overhead for changing the number of virtual machines running in the node, $\forall i \in \mathscr{V}(t)$, $\Delta_i = \Delta$. This means that the optimization problem for assigning the node deadlines is given by (8.24). Naturally, the node deadlines will deviate from this whenever there is a transition towards a new goal point $\mathbf{D}^*$. However, the new goal point will be chosen according to the optimization problem (but obviously with the added constraint of the new flow).

$$\begin{aligned} \text{minimize} \quad & \sum_{i \in \mathscr{V}(t)} 1/D_i \\ \text{subject to} \quad & \sum_{i=1}^{\ell_j} (1+\alpha)^{\ell_j - i} D_{p_j(i)} \leq \mathscr{D}_j \quad && \forall j \in \mathscr{F}(t) \\ & D_i \geq 0 && \forall i \in \mathscr{V}(t) \end{aligned} \tag{8.24}$$

Evaluation methodology To evaluate how the choice of α affects the time required to accept a new flow, we will evaluate how long it takes the system presented earlier to transition between consecutive goal points $\mathbf{D}^* \in \mathbb{D}(\mathscr{G}^+)$. To do this, we used the following methodology:

1. Choose the order-of-magnitude for the end-to-end deadlines in the network $\overline{\mathscr{D}} \in \{0.1,\ 1,\ 10\}$ ms.

2. Choose a value for $\alpha \in [10^{-3},\ 10^{0}]$.

3. Assign a randomly generated end-to-end deadline to each flow drawn from a uniform distribution, $\mathscr{D}_j \in \mathscr{U}(0.7 \cdot \overline{\mathscr{D}},\ 1.3 \cdot \overline{\mathscr{D}})$.

4. Chose a new *goal point* $\mathbf{D}^*$ as the solution to the optimization problem (8.24).

5. Simulate how long it takes the network to reach the desired goal point $\mathbf{D}^*$.

6. Repeated steps 3 thru 4 for 100 simulations.

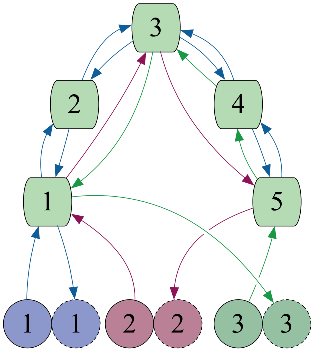


Figure 8.9 Illustration of the network used to evaluate the trade-offs with α and the time taken to reach the goal point $\mathbf{D}^*$.

7. From the 100 simulations, compute the *average time* to reach a goal point $\mathbf{D}^*$.
8. Repeat steps 1 thru 7 for a different choices of $\overline{\mathscr{D}}$ and α.

Results The result of the evaluation is shown in Figure 8.10. As expected, it shows a clear relationship between α and the time needed to reach $\mathbf{D}^*$. It is interesting to note that even when having end-to-end deadlines in the order of 10ms, and a very small α the resource manager is still able to reach $\mathbf{D}^*$ within a second. By allowing a higher α, it is possible to reach $\mathbf{D}^*$ in less than a microsecond.

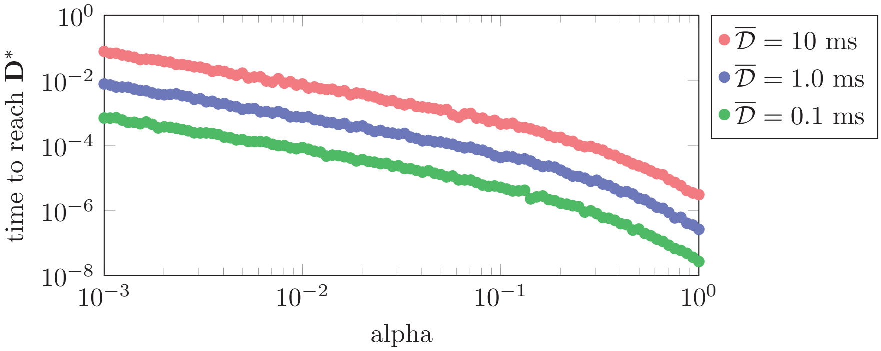


Figure 8.10 Simulation result of how long it takes a system (with the network depicted in Figure 8.9) to reach a goal-point $\mathbf{D}^*$. It shows how this time depends on both the choice of α but also on how large the end-to-end deadlines of the flows are.

8.5 Summary

In this chapter, we presented a framework, which allows applications and cloud-services to dynamically join and leave a system over time. The intuition is that by assigning and controlling how quickly local deadlines of the node may change, it is possible to guarantee the end-to-end deadlines of the applications in presence of flows and nodes dynamically leaving and joining the network. Finally, with extensive simulations we are able to show that with the suggested protocols it is possible to accept new applications in a matter of milliseconds.

Regarding the choice of bounding how quickly the individual node deadlines are allowed to change over time, i.e., bounding $|\dot{D}_i(t)| \leq \alpha$, one should note that it is sufficient to only bound the positive derivative. Theoretically, the theorems presented in this chapter still hold. This means that one could theoretically lower the node deadlines in a step-wise fashion, something that would allow the system to accept new flows immediately. The problem with this, however, is that the nodes might not be able to lower their response-time in a step-wise fashion. But if one would allow the nodes to state a bound on how quickly they could lower their response-time, it could theoretically allow the system accept new flows faster. Since the rate-of-change for lowering the node deadlines would be bounded by that bound, rather than by alpha.

Finally, this work opens up for a variety of directions for future work. For instance different ways to do response-time control, as well as how the proposed methodology scales with the number of nodes and flows in a network.

9

Conclusions and future work

In this thesis we have progressively ventured towards the next-generation of cloud systems. The goal has been to be able to guarantee that traffic passing through a set of cloud services, or virtual network functions, can do so within a specific end-to-end deadline. The purpose of this thesis was to present one way to model a real-time cloud in order to achieve this goal. The focus was on the holistic analysis and to illustrate how one can combine ideas from control theory, real-time systems, and network calculus to connect, what we believe to be, the fundamental building blocks needed to achieve this goal. We believe these building blocks to be:

- Deadlines for traffic passing through the network;
    - This includes both local node deadlines as well as end-to-end deadlines for flows passing through multiple nodes.
- Scaling of processing capacity;
    - Controlling the processing capacity of a node by scaling the number of virtual machines (horizontal scaling) or the size of a virtual machine (vertical scaling) to ensure that the traffic passing through the node meets a specific deadline.
- Admission control;
    - Controlling the rate by which traffic is admitted into a node, such that the response-time of a node is kept below a specific deadline.
- Control of the local deadlines of the nodes;
    - Controlling the local deadlines of the nodes in the network in a dynamic way so that we can ensure that the end-to-end deadlines for the traffic flows in the networks are met.

Stimulate future research The goal when illustrating ways to combine these fundamental building blocks is to stimulate further research within this area, so that the policies and methods proposed in this thesis can be replaced with improved and better versions. For instance, in a real system, the behavior of the incoming traffic could be more complicated, such that using linear extrapolation to predict the future arrival rate will no longer be sufficient. Instead, one might need to use machine learning in order to do such predictions. However, with the proposed abstraction of AutoSAC, one can still feed this prediction into the proposed scaling-policy.

In a similar way, in a real system it might be necessary to use more advanced ways to predict the future performance of the virtual machines running in a node. The method proposed in this thesis might no longer be sufficient. For instance, the processing capacity of a virtual network function could depend on a myriad of things, such as the type of traffic, the size of the incoming packets, the size of the routing tables within the node, how many internal queues are in the node, etc. However, if one uses more advanced ways to predict the future processing capacity of the node, one can still feed this into the admission controller and the service controller proposed in this thesis.

Controlling a single node

In the first two technical chapters of this thesis, i.e., Chapters 3 and 4, we present two different ways to control a cloud node. The problem of controlling a single cloud node might not be very interesting in itself, but the purpose of these two chapters was to introduce the reader to the fundamental problems and building blocks studied in this thesis. One of these problems is that it becomes difficult to provide deadline-guarantees when the time needed to change the processing capacity of a node is larger than this deadline. Should the time-overhead to change the processing capacity be less than the deadline, then the problem becomes much simpler, and would allow the system to use a more reactive approach. Another problem is that with horizontal scaling, one has only coarse-grained control over the processing capacity of the node. It therefore becomes difficult to exactly match the required processing capacity. Recall that using processing capacity was assumed to incur a cost, so one wish to guarantee the deadline while using as few resources as possible. In a perfect world, one would have smooth control over the processing capacity, and no time-overhead to change it. In such a world, the problems of guaranteeing that the response-time of the node was below a deadline would be almost trivial.

To address these problems, we propose a way to combine admission control with horizontal scaling, in Chapter 3. This allows us to ensure that the response-time of a node remains below a specific deadline. A downside of this approach, however, is that to ensure that the response-time is below the deadline, the node sometimes have to discard some of the incoming packets. This is not ideal, but one can view it as the job of the service controller to ensure that there is sufficient processing capacity in the node, so that the admission controller does not have to discard any packets.

This methodology used in AutoSAC is not mutually exclusive with HoloScale, the one presented in Chapter 4. In fact, the cascaded control-structure proposed in HoloScale, can be combined with AutoSAC. The idea is that the admission controller in AutoSAC should act as the quality-of-service-controller in HoloScale, and that the HoloScale-controller can be used as the inner controller. Unfortunately, the combination of these two methodologies was never explored in this thesis, but is something that would be interesting to explore in future work.

A matter of time-scales It might seem that the time-scales used to model the cloud systems in this thesis are quite conservative. What would happen if there was a reduction in the time-overhead needed to start a new virtual machine? In such a scenario, one could argue that the system would be able to handle even smaller deadlines, and more volatile traffic. By changing the time-overhead from 1 s to 1 ms, we could change the deadlines from 1 ms to 1 μs, and still preserve the difference of the time-scales exploited when deriving AutoSAC. Therefore, it is possible that the theory developed in this thesis remains valid even after technological breakthroughs in virtualization techniques.

Controlling a chain of nodes

In the middle-two technical chapters of the thesis we studied the problem of controlling a chain of nodes. The goal was to ensure that the end-to-end response time for the chain remained below a given end-to-end deadline. This meant that packets should not spend more than the given end-to-end deadline to pass through all the nodes in the chain. Along with the problems of controlling a single node, this added a new problem: the interaction between nodes. For instance, if one node needed more time to process packets, it would leave less time for subsequent nodes to process packets.

In Chapter 5 we proposed a way for the nodes to do horizontal scaling in order to achieve these goals. This resulted in every node following a specific schedule for turning on and off a virtual machine. Notably, every node in the chain followed a schedule with the same period, which is a quite restrictive constraint. Note however, that different nodes could still have their extra virtual machine on for different length of time. The main benefit of this approach however, was that it did not discard any packets.

To get away from the approach of forcing every node to follow the same period, we proposed to control the nodes using the AutoSAC-methodology. This allowed the system to handle more dynamic traffic, but with the downside of assigning a local node deadline to every node in the chain. This meant, for example, that a packet could be processed by the first three nodes and then be dropped by the fourth one. It would be interesting to compare the use of an admission controller in every node with having a single admission controller for the entire chain. It is not obvious which of the two approaches would be best.

Controlling a network of nodes

In the final two technical chapters, Chapters 7 and 8 we added another fundamental problem: the interaction between multiple flows. In essence, this means that multiple flows pass through the networks, and they might affect each other. In Chapter 7 we extended AutoSAC and proposed one way to address this. The main idea is to use more communication between the nodes in order to achieve a better prediction of the traffic flowing through the network. We also proposed a way to assign the local node deadlines such that the end-to-end deadlines of the flows traversing the network would not be violated.

Why not an admission controller in the beginning of each path? Going back to the question of having a single admission controller per flow, one can again ask this question for a network of nodes. This is interesting, because in this case it is not possible without an upper bound on the arriving traffic of the flows. For instance, let us consider a chain of nodes, but where each node is also affected by a cross-flow. To be able to ensure that the packets admitted into the flow are guaranteed to meet the end-to-end deadline, we would need an upper bound on the traffic flow for all the cross-flows passing through the nodes. The reason for this is because knowing the states of the system when admitting the packets to the first node, is not sufficient to provide an upper bound on the end-to-end latency of the entire chain. For instance, the node could traverse five nodes, and then while being in the sixth node, the cross-flow of node seven has a burst of traffic, rapidly increasing the response-time of this node. In such a scenario, the response-time of the seventh node might be too large to ensure that the packet belonging to the flow will meet its end-to-end deadline.

Dynamically changing the node deadlines One artifact from AutoSAC is that its admission controller allows the nodes to dynamically change the node deadline in a continuous and smooth way and still guarantee that the response-time remains below this deadline. This is what sparked the idea behind Chapter 8. One could for instance imagine that something happens in the system, and that one node would like to increase its node deadline in order to handle a sudden burst of incoming traffic. However, doing so could affect the end-to-end response times of the flows, and might require us to change the local deadlines of the other nodes in the network. One way to address this is to use the methodology presented in Chapter 8.

Scalability of the network Both methods proposed in the network-chapters of this thesis beg the question of how they scale with an increase of the number of nodes and flows in the network. This has not been explored within the thesis. It would therefore be interesting to investigate how one can combine graph-based deep learning and network calculus in order to dynamically dimension the processing capacity of the nodes in a very large network. This idea is inspired by similar work that was explored by Geyer [Geyer and Bondorf, 2019].

9.1 Future work – a dynamic network calculus

As hinted in the discussion around whether one should use a single admission controller for each flow, or one admission controller for every node, it would be very interesting to study what one can do if one has a model of the upper bound on the traffic flowing through the nodes. This is something quite common in the field of network calculus, and it would be very interesting to study what kind of performance and guarantees one can achieve when using the network calculus theory in a more dynamic way.

Recall from Chapter 1 that a common way to model an upper bound on the arriving traffic is with *arrival curves*. One example is the burst-rate model, which is modeled by a quick burst of traffic, followed by a steady-state rate:

$$\alpha(t) = b + r \cdot t,$$

where b is the initial burst-size and r is the steady-state rate of the traffic.

Similar to the arrival curves, the lower bound of the available processing capacity of a node can be modeled using a *service curve*. One example of a service curve is the latency-rate model:

$$\beta(t) = [C \cdot (t - L)]^{+},$$

where C is the processing rate of the server, and L is the latency,

Dynamically arrival and service curves

One idea for future work would be to allow the service curves and arrival curves to dynamically change over time. One would then use them to control the processing capacity over time, or the admission control. For instance, one could use the service and arrival curves to scale the processing capacity of a node such that the number of packets being discarded in a node is bounded.

Scaling a node with fixed buffer size Assuming that every machine-instance in the node has a fixed queue-size of q_k^{max}, one can use the service and arrival curves to find the minimum amount of instances needed in order to guarantee that the node never drops more than a fraction ε of the incoming packets. The assumption is that if the queue is full, and the node has incoming traffic, it will have to drop any incoming traffic until there is room in its queue. To ensure that no more than a fraction ε of the incoming traffic is dropped, the node can scale the number of horizontal machine according to Equation (9.1):

$$m(t) = \max\left(\lceil (1-\varepsilon) \cdot b / q_k^{\mathsf{max}} \rceil,\ \lceil (1-\varepsilon) \cdot r / \bar{s} \rceil\right) \tag{9.1}$$

where $\bar{s}$ is the maximum packet-processing rate of a single machine instance, where q_k^{max} is the maximum queue-size of a single virtual machine instance, and finally where b and r are the burstiness and the intensity of the arriving traffic, respectively.

Scaling a node with fixed deadline Similar to before, it is possible to derive a scaling policy to ensure that a node never drops more than a fraction ε of the incoming traffic when the node has a fixed deadline. Again, the underlying assumption is that the node has an admission controller that will drop any incoming traffic whenever the response-time of the node is larger than this deadline, just like the one found in AutoSAC.

In this case, one can ensure that the no more than a fraction ε of the incoming traffic will be dropped, as long as the horizontal scaling is performed according to Equation (9.2):

$$m(t) = \max\left(\left\lceil (1-\varepsilon)\cdot\frac{b}{\bar{s}\cdot\mathscr{D}}\right\rceil, \left\lceil (1-\varepsilon)\cdot\frac{r}{\bar{s}}\right\rceil\right), \tag{9.2}$$

where $\mathscr{D}$ is the deadline for the packets passing through the node. The intuition behind this policy, which can be seen in Figure 9.1, is that we have to choose the number of virtual machines such that the dashed blue line is above the end of the red line, which is indicated the given deadline.

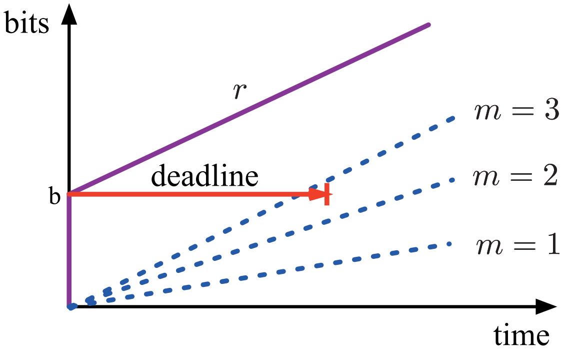


Figure 9.1 Illustration behind the scaling policy of Equation (9.2). The idea is to choose the number of instances such that the dashed blue line is above the end of the red line, which indicated the deadline.

Dynamic priority assignment to flows of a node Another interesting direction for future work is to use network calculus models to dynamically assign priorities to the flows passing through a node. In Chapters 5 and 6 we did not distinguish between flows. However, it might be useful to prioritize the traffic belonging to different flows based on their *laxity*, which is the time left before the deadline is up. The laxity for a packet entering node i and belonging to flow j is given by

$$\mathrm{lax}_{i,j}(t) = \mathscr{D}_j - \mathscr{L}_{i,j}(t),$$

where $\mathscr{D}_j$ is the end-to-end deadline of flow j and $\mathscr{L}_{i,j}(t)$ is the time it took the packet entering node i to get there. Therefore, $\mathrm{lax}_{i,j}(t)$ is the time the packet has left, before its end-to-end deadline is met. The idea is then to assign priorities to the flows passing through a node i according to the *least-laxity-first* principle, meaning that the flow with the least laxity gets the highest priority of a flow, and the flow with the largest laxity gets the lowest priority.

One can also combine this priority-assignment policy with the scaling policy presented in Equation (9.2), but where one uses the laxity instead of the deadline. This might allow us to have a completely decentralized scaling policy, and priority policy yet still guarantee that the end-to-end deadline of every flow in the network is always met.

Summary

There are many interesting directions in which one can combine network calculus and use it in more dynamic ways, as illustrated above. A downside, however, is that when dynamically scaling the number of virtual machines, for instance as proposed in Equation (9.2), or when doing the dynamic priority assignment, it might be difficult to provide hard guarantees. However, this might be an acceptable trade-off since it will allow the system to adapt and react to changes during run-time.

Another caveat to consider when using network calculus is how good it actually is for modeling the performance of virtual network functions, or cloud services. Their behavior can be quite complicated and change depending on a myriad of things, such as the type of incoming traffic, the size of the packets, or the size of the routing table inside the nodes.

Bibliography

Adhikari, V. K., Y. Guo, F. Hao, M. Varvello, V. Hilt, M. Steiner, and Z.-L. Zhang (2012). “Unreeling Netflix: Understanding and Improving Multi-CDN Movie Delivery”. In: *IEEE Conference on Computer Communications (INFOCOM)*. IEEE, pp. 1620–1628.

Aijaz, A., M. Dohler, A. H. Aghvami, V. Friderikos, and M. Frodigh (2016). “Realizing the Tactile Internet: Haptic Communications over next Generation 5G Cellular Networks”. *IEEE Wireless Communications* **24**:2, pp. 82–89.

Armbrust, M, A Fox, R Griffith, A. D. Joseph, R. H. Katz, A Konwinski, G Lee, D. A. Patterson, A Rabkin, I Stoica, and M Zaharia (2009). “Above the Clouds: A Berkeley View of Cloud Computing”. *University of California, at Berkeley, Technical Report No. UCB/EECS-2009-28*.

Ashjaei, M., S. Mubeen, M. Behnam, L. Almeida, and T. Nolte (2016). “End-to-End Resource Reservations in Distributed Embedded Systems”. In: *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 1–11.

Åström, K. J. and R. M. Murray (2010). *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press.

Baccelli, F., G. Cohen, G. J. Olsder, and J.-P. Quadrat (1992). *Synchronization and Linearity*. Vol. 3. Wiley New York.

Basiri, A., N. Behnam, R. De Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal (2016). “Chaos Engineering”. *IEEE Software* **33**:3, pp. 35–41.

Bengtsson, K. (2017). *Lessoned Learned Implementing "Industry 4.0" at Volvo Cars*. LCCC Seminar at Lund University.

Beyer, B., C. Jones, J. Petoff, and N. R. Murphy (2016). *Site Reliability Engineering: How Google Runs Production Systems*. " O’Reilly Media, Inc.".

Bonafiglia, R., I. Cerrato, F. Ciaccia, M. Nemirovsky, and F. Risso (2015). “Assessing the Performance of Virtualization Technologies for NFV: A Preliminary Benchmarking”. In: *Fourth European Workshop on Software Defined Networks*. IEEE, pp. 67–72.

Bouillard, A., M. Boyer, and E. Le Corronc (2018). *Deterministic Network Calculus: From Theory to Practical Implementation*. John Wiley & Sons.

Boyd, S. and L. Vandenberghe (2004). *Convex Optimization*. Cambridge University Press.

Chakraborty, S. and L. Thiele (2005). “A New Task Model for Streaming Applications and its Schedulability Analysis”. In: *Design, Automation and Test in Europe Conference and Exposition*, pp. 486–491.

Chang, M. A., B. Tschaen, T. Benson, and L. Vanbever (2015). “Chaos Monkey: Increasing SDN Reliability through Systematic Network Destruction”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 45. 4. ACM, pp. 371–372.

Cockroft, A., C. Hicks, and G. Orzell (2011). “Lessons Netflix learned from the AWS outage”. *Netflix Techblog*.

Cohen, R., L. Lewin-Eytan, J. S. Naor, and D. Raz (2015). “Near Optimal Placement of Virtual Network Functions”. In: *IEEE Conference on Computer Communications (INFOCOM)*. IEEE, pp. 1346–1354.

Cruz, R. L. (1991a). “A Calculus for Network Delay. I. Network Elements in Isolation”. *IEEE Transactions on information theory* **37**:1, pp. 114–131.

Cruz, R. L. (1991b). “A Calculus for Network Delay. II. Network Analysis”. *IEEE Transactions on Information Theory* **37**:1, pp. 132–141.

Di Natale, M. and J. A. Stankovic (1994). “Dynamic end-to-end guarantees in distributed real time systems”. In: *Proceedings of the 15th IEEE Real-Time Systems Symposium (RTSS)*, pp. 215–227.

ETSI (2012). *Network Functions Virtualization (NFV)*. URL: `https://portal.etsi.org/nfv/nfv_white_paper.pdf`.

ETSI (2013). *Network Functions Virtualization (NFV); Use Cases*. URL: `https://www.etsi.org/deliver/etsi_gs/NFV/001_099/001/01.01.01_60/gs_NFV001v010101p.pdf`.

Faraci, G., A. Lombardo, and G. Schembra (2017). “A building block to model an SDN/NFV network”. In: *IEEE International Conference on Communications (ICC)*. IEEE, pp. 1–7.

Feng, H., J. Llorca, A. M. Tulino, and A. F. Molisch (2016). “Dynamic Network Service Optimization in Distributed Cloud Networks”. In: *IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, pp. 300–305.

Fettweis, G. P. (2014). “The Tactile Internet: Applications and Challenges”. *IEEE Vehicular Technology Magazine* **9**:1, pp. 64–70.

Gandhi, N., D. M. Tilbury, Y. Diao, J Hellerstein, and S. Parekh (2002). “MIMO Control of an Apache Web Server: Modeling and Controller Design”. In: *Proceedings of the 2002 American Control Conference*. Vol. 6. IEEE, pp. 4922–4927.

Gerber, R., S. Hong, and M. Saksena (1995). “Guaranteeing Real-Time Requirements with Resource-based Calibration of Periodic Processes”. *IEEE Transaction on Software Engineering* **21**:7, pp. 579–592.

Geyer, F. and S. Bondorf (2019). “DeepTMA: Predicting Effective Contention Models for Network Calculus using Graph Neural Networks”. In: *IEEE Conference on Computer Communications (INFOCOM)*. IEEE, pp. 1009–1017.

Hamann, A., M. Jersak, K. Richter, and R. Ernst (2006). “A Framework for Modular Analysis and Exploration of Heterogeneous Embedded Systems”. *Real-Time Systems* **33**:1, pp. 101–137.

Henriksson, D., Y. Lu, and T. Abdelzaher (2004). “Improved Prediction for Web Server Delay Control”. In: *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 61–68.

Hong, S., T. Chantem, and X. S. Hu (2015). “Local-Deadline Assignment for Distributed Real-Time Systems”. *IEEE Transactions on Computers* **64**:7, pp. 1983–1997.

Jacob, R., M. Zimmerling, P. Huang, J. Beutel, and L. Thiele (2016). “End-to-End Real-Time Guarantees in Wireless Cyber-Physical Systems”. In: *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*. IEEE, pp. 167–178.

Jayachandran, P. and T. Abdelzaher (2008). “Delay Composition Algebra: A Reduction-Based Schedulability Algebra for Distributed Real-Time Systems”. In: *Proceedings of the 29th IEEE Real-Time Systems Symposium (RTSS)*, pp. 259–269.

Jiang, S. (2006). “A Decoupled Scheduling Approach for Distributed Real-Time Embedded Automotive Systems”. In: *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 191–198.

Jiang, Y. and Y. Liu (2008). *Stochastic Network Calculus*. Vol. 1. Springer.

Jonas, E., J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson (2019). “Cloud Programming Simplified: A Berkeley View on Serverless Computing”. *arXiv.org*. arXiv: `1902.03383v1`.

Kapoor, R., G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat (2012). “Chronos: Predictable Low Latency for Data Center Applications”. In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC ’12. ACM, San Jose, California, 9:1–9:14.

Klein, C., M. Maggio, K.-E. Årzén, and F. Hernández-Rodriguez (2014). “Brownout: Building More Robust Cloud Applications”. In: *Proceedings of the 36th International Conference on Software Engineering*. ACM, pp. 700–711.

Kleinrock, L. (1975). *Queueing Systems*. John Wiley & Sons.

Kuo, T. W., B. H. Liou, K. C. J. Lin, and M. J. Tsai (2016). “Deploying Chains of Virtual Network Functions: On the Relation Between Link and Server Usage”. In: *IEEE International Conference on Computer Communications (INFOCOM)*, pp. 1–9.

Le Boudec, J.-Y. and P. Thiran (2001). *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Vol. 2050. Lecture Notes in Computer Science. Springer.

Leitner, P. and J. Cito (2016). “Patterns in the Chaos—A Study of Performance Variation and Predictability in Public IaaS Clouds”. *ACM Transactions on Internet Technology* **16**:3, pp. 1–23.

Leivadeas, A., M. Falkner, I. Lambadaris, and G. Kesidis (2016). “Resource Management and Orchestration for a Dynamic Service Chain Steering Model”. In: *IEEE Global Communications Conference (GLOBECOM)*. IEEE, pp. 1–6.

Li, X. and C. Qian (2015). “Low-Complexity Multi-Resource Packet Scheduling for Network Function Virtualization”. In: *IEEE Conference on Computer Communications (INFOCOM)*, pp. 1400–1408.

Li, Y., L. Phan, and B. T. Loo (2016). “Network Functions Virtualization with Soft Real-Time Guarantees”. In: *IEEE International Conference on Computer Communications (INFOCOM)*. IEEE. IEEE.

Lin, T., Z. Zhou, M. Tornatore, and B. Mukherjee (2014). “Optimal Network Function Virtualization Realizing End-to-End Requests”. In: *IEEE Global Communications Conference (GLOBECOM)*. IEEE, pp. 1–6.

Liu, F., J. Tong, J. Mao, R. Bohn, J. Messina, L. Badger, and D. Leaf (2011). “NIST Cloud Computing Reference Architecture”. *NIST Special Publication* **500**:2011, pp. 1–28.

Lorente, J. L., G. Lipari, and E. Bini (2006). “A Hierarchical Scheduling Model for Component-Based Real-Time Systems”. In: *Proceedings of the 20-th International Parallel and Distributed Processing Symposium*. Rhodes Island, Greece.

Mao, M., J. Li, and M. Humphrey (2010). “Cloud Auto-Scaling with Deadline and Budget Constraints”. In: *11th IEEE/ACM International Conference on Grid Computing*. IEEE, pp. 41–48.

Marinca, D., P. Minet, and L. George (2004). “Analysis of Deadline Assignment Methods in Distributed Real-Time Systems”. *Computer Communications* **27**:15, pp. 1412–1423.

Mehraghdam, S., M. Keller, and H. Karl (2014). “Specifying and Placing Chains of Virtual Network Functions”. In: *3rd IEEE International Conference on Cloud Networking (CloudNet)*. IEEE, pp. 7–13.

Mell, P., T. Grance, et al. (2011). “The NIST definition of cloud computing”.

Millnert, V. (2014). *Quality-Latency Trade-Off in Bilateral Teleoperation*. MSc. thesis. Department of Automatic Control, Lund University.

Millnert, V., J. Eker, and E. Bini (2017). "Dynamic Control of NFV Forwarding Graphs with End-to-End Deadline Constraints". In: *IEEE International Conference on Communications*. IEEE. Paris, France, pp. 1–7.

Millnert, V., J. Eker, and E. Bini (2018). "Achieving Predictable and Low End-to-End Latency for a Network of Smart Services". In: *IEEE Global Communications Conference*. IEEE. IEEE, Abu Dhabi, UAE, pp. 1–7.

Moens, H. and F. De Turck (2014). "VNF-P: A Model for Efficient Placement of Virtualized Network Functions". In: *10th International Conference on Network and Service Management (CNSM)*. IEEE, pp. 418–423.

Nylander, T., M. T. Andrén, K.-E. Årzén, and M. Maggio (2018). "Cloud Application Predictability through Integrated Load-Balancing and Service Time Control". In: *IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, pp. 51–60.

Nyquist, H. (1932). "Regeneration Theory". *Bell System Technical Journal* **11**:1, pp. 126–147.

Ousterhout, K., P. Wendell, M. Zaharia, and I. Stoica (2013). "Sparrow: Distributed, Low Latency Scheduling". In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. ACM, Farminton, Pennsylvania, pp. 69–84.

Padala, P., K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem (2007). "Adaptive Control of Virtualized Resources in Utility Computing Environments". In: *ACM Operating Systems Review (SIGOPS)*. Vol. 41. 3. ACM, pp. 289–302.

Palencia, J. and M. G. Harbour (2003). "Offset-Based Response Time Analysis of Distributed Systems Scheduled Under EDF". In: *15th Euromicro Conference on Real-Time Systems (ECRTS)*. Porto, Portugal.

Parekh, A. K. and R. G. Gallager (1993). "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: the Single-Node Case". *IEEE/ACM Transactions on Networking* **1**:3, pp. 344–357.

Pellizzoni, R. and G. Lipari (2007). "Holistic Analysis of aSynchronous Real-Time Transactions with Earliest Deadline Scheduling". *Journal of Computer and System Sciences* **73**:2, pp. 186–206.

Rahni, A., E. Grolleau, and M. Richard (2008). "Feasibility Analysis of Non-Concrete Real-Time Transactions with EDF Assignment Priority". In: *Proceedings of the 16th conference on Real-Time and Network Systems*. Rennes, France, pp. 109–117.

Ren, Y., T. Phung-Duc, J.-C. Chen, and Z.-W. Yu (2016). "Dynamic Auto Scaling Algorithm (DASA) for 5G Mobile Networks". In: *IEEE Global Communications Conference (GLOBECOM)*. IEEE, pp. 1–6.

Serreli, N., G. Lipari, and E. Bini (2009). "Deadline Assignment for Component-Based Analysis of Real-Time Transactions". In: *2nd Workshop on Compositional Real-Time Systems*. Washington, DC, USA.

Shen, W., M. Yoshida, T. Kawabata, K. Minato, and W. Imajuku (2014). "vConductor: An NFV Management Solution for Realizing End-to-End Virtual Network Services". In: *16th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE, pp. 1–6.

Simsek, M., A. Aijaz, M. Dohler, J. Sachs, and G. Fettweis (2016). "5G-enabled Tactile Internet". *IEEE Journal on Selected Areas in Communications* **34**:3, pp. 460–473.

Thönes, J. (2015). "Microservices". *IEEE Software* **32**:1, pp. 116–116.

Tindell, K. and J. Clark (1994). "Holistic Schedulability Analysis for Distributed Hard Real-Time Systems". *Microprocessing and Microprogramming* **50**, pp. 117–134.

Tindell, K. W., A. Burns, and A. Wellings (1994). "An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks". *Journal of Real Time Systems* **6**:2, pp. 133–152.

Voigt, T. and P. Gunningberg (2002). "Adaptive Resource-Based Web Server Admission Control". In: *ISCC*.

Wang, X., C. Wu, F. Le, A. Liu, Z. Li, and F. Lau (2016). "Online VNF Scaling in Datacenters". In: *IEEE 9th International Conference on Cloud Computing (CLOUD)*. IEEE, pp. 140–147.

Printed by Media-Tryck, Lund 2019 NORDIC SWAN ECOLABEL 3041 0903

LUND UNIVERSITY

Department of Automatic Control
P.O. Box 118, 221 00 Lund, Sweden
www.control.lth.se

PhD Thesis TFRT-1126
ISBN 978-91-7895-235-9
ISSN 0280–5316